

CAPÍTULO 9.

Almacenamiento de datos

Las aplicaciones descritas hasta este capítulo representan la información a procesar en forma de variables. El problema con las variables es que dejan de existir en el momento en que la aplicación es destruida. En muchas ocasiones vamos a necesitar almacenar información de manera permanente. Las alternativas más habituales para conservar esta información son los ficheros, las bases de datos o servicios a través de la red. Estas técnicas no solo permiten mantener a buen recaudo los datos de la aplicación, si no que también permiten compartir estos datos con otras aplicaciones y usuarios. De forma adicional, el sistema Android pone a nuestra disposición dos nuevos mecanismos para almacenar datos, las preferencias y ContentProvider.

A lo largo de este capítulo estudiaremos cómo utilizar estas técnicas en Android. Comenzaremos describiendo el uso de las preferencias como un mecanismo sencillo para guardar de forma permanente algunas variables. Seguiremos describiendo las características del sistema de ficheros que incorpora Android. Se puede acceder a los ficheros a través de las clases estándar incluidas en Java. De forma adicional se incluyen nuevas clases para cubrir las peculiaridades de Android.

Como tercera alternativa se estudiará el uso de XML para almacenar la información de manera estructurada. Similar a XML tenemos JSON, que presenta la ventaja de usar un formato más compacto. Como quinta alternativa al almacenamiento de datos se estudiarán las bases de datos. Android incorpora la librería SQLite, que nos permitirá crear y manipular nuestras propias bases de datos de forma muy sencilla. Para finalizar, se describirá la clase `ContentProvider`, que consiste en un mecanismo introducido en Android para poder compartir datos entre aplicaciones.

En el capítulo siguiente se describe otra alternativa, el uso de Internet como recurso para almacenar y compartir información. Concretamente se describirá el uso de *sockets* TCP, HTML y los servicios web.

Todas estas alternativas se ilustrarán a través del mismo ejemplo. Trataremos de almacenar la lista con las mejores puntuaciones obtenidas en Asteroides, tal como se ha descrito en el capítulo 3.



Objetivos:

- Repasar las alternativas para el almacenamiento de datos en Android.
- Describir el uso de ficheros.
- Utilizar dos herramientas para manipular ficheros XML (SAX y DOM) y dos herramientas para manipular ficheros JSON (GSON y org.json).
- Mostrar como desde Android podemos utilizar SQLite para trabajar con bases de datos relacionales.
- Describir qué es un ContentProvider y cómo podemos utilizar algunos ContentProvider disponibles en Android.
- Aprender a crear nuestros propios ContentProvider.

9.1. Alternativas para guardar datos permanentemente en Android

Existen muchas alternativas para almacenar información de forma permanente en un sistema informático. A continuación, mostramos una lista de las más habituales utilizadas en Android:

- **Preferencias:** Es un mecanismo liviano que permite almacenar y recuperar datos primitivos en forma de pares clave/valor. Este mecanismo se suele utilizar para almacenar los parámetros de configuración de una aplicación.
- **Ficheros:** Puedes almacenar los ficheros en la memoria interna del dispositivo o en un medio de almacenamiento externo, como una tarjeta SD. También puedes utilizar ficheros añadidos a tu aplicación, como recursos.
- **XML:** Se trata de un estándar fundamental para la representación de datos, en Internet y en muchos otros entornos (como en el Android SDK). En Android disponemos de las librerías SAX y DOM para manipular datos en XML.
- **JSON:** Es una alternativa a XML para almacenar información estructurada. Usa una representación simple y compacta, lo que la hace especialmente interesante para transacciones por Internet. En este capítulo se describen dos herramientas: GSON y org.json.
- **Base de datos:** Las API de Android contienen soporte para SQLite. Tu aplicación puede crear y usar bases de datos SQLite de forma muy sencilla y con toda la potencia que nos da el lenguaje SQL.
- **Proveedores de contenido:** Un proveedor de contenido es un componente de una aplicación que expone el acceso de lectura/escritura de sus datos a otras aplicaciones. Está sujeto a las restricciones de seguridad que quieras imponer. Los proveedores de contenido implementan una sintaxis estándar para acceder a sus datos mediante URI (Uniform Resource Identifiers) y un mecanismo de acceso para devolver los datos similar a SQL. Android provee algunos proveedores de contenido para tipos de datos estándar, tales como contactos personales, ficheros multimedia, etc.
- **Internet:** No te olvides de que también puedes usar la nube para almacenar y recuperar datos. Se estudia en el siguiente capítulo.



Vídeo[tutorial]: *Almacenamiento de datos en Android*

9.2. Añadiendo puntuaciones en Asteroides

A modo de ejemplo se va a implementar la posibilidad de guardar las mejores puntuaciones obtenidas en Asteroides. Se utilizarán mecanismos alternativos que se desarrollarán a lo largo de este capítulo y el siguiente:

- Array (implementado en el capítulo 3)
- Preferencias
- Ficheros en memoria interna, externa y en recursos
- XML con SAX y DOM
- JSON con GSON y org.json
- Base de datos SQLite y con varias tablas relacionales
- ContentProvider
- Internet a través de *sockets*
- Servicios web

Para facilitar la sustitución del método de almacenamiento, en el capítulo 3 se ha creado la siguiente interfaz en la aplicación Asteroides:

```
public interface AlmacenPuntuaciones {
    public void guardarPuntuacion(int puntos, String nombre, long fecha);
    public List<String> listaPuntuaciones(int cantidad);
}
```

También se ha declarado la variable `almacen` de tipo `AlmacenPuntuaciones` y se ha creado la actividad `Puntuaciones`, que visualiza un `ListView` con las puntuaciones. Dado que en el capítulo 3 todavía no teníamos la opción de jugar, no se podían añadir nuevas puntuaciones a `almacen`. En el siguiente ejercicio trataremos de calcular una puntuación en el juego y almacenarla en `almacen`.



Ejercicio: Calculando la puntuación en Asteroides

1. Crea una variable global en la clase `VistaJuego` que se llame `puntuacion` e inicialízala a cero:

```
private int puntuacion = 0;
```

2. Cada vez que se destruya un asteroide hay que incrementar esta variable. Añade dentro de `destruyeAsteroide()` la siguiente línea:

```
puntuacion += 1000;
```

3. Cuando desde la actividad inicial `Asteroides` se llame a la actividad `Juego`, nos interesa que esta nos devuelva la puntuación obtenida. Recuerda que en el capítulo 3 hemos estudiado la comunicación entre actividades. Para pasar la información entre las actividades, añade el siguiente código en `Asteroides` en sustitución del método `lanzarJuego()` anterior:

```
static final int ACTIV_JUEGO = 0;

public void lanzarJuego(View view) {
    Intent i = new Intent(this, Juego.class);
    startActivityForResult(i, ACTIV_JUEGO);
}

@Override protected void onActivityResult (int requestCode,
                                             int resultCode, Intent data){
    super.onActivityResult(requestCode, resultCode, data);
    if (requestCode== ACTIV_JUEGO && resultCode==RESULT_OK && data!=null) {
        int puntuacion = data.getExtras().getInt("puntuacion");
        String nombre = "Yo";
        // Mejor leer nombre desde un AlertDialog.Builder o preferencias
        almacen.guardarPuntuacion(puntuacion, nombre,
                                   System.currentTimeMillis());
        lanzarPuntuaciones(null);
    }
}
```

4. Para realizar la respuesta de la actividad será más sencillo hacerlo desde `VistaJuego` que desde `Juego`. El problema es que esta clase es una vista, no una actividad. Para solucionar el problema puedes usar el siguiente truco. Introduce en `VistaJuego` el siguiente código:

```
private Activity padre;

public void setPadre(Activity padre) {
    this.padre = padre;
}
```

5. Cuando se detecte una condición de victoria o derrota, es un buen momento para almacenar la puntuación y salir de la actividad. Para ello crea el siguiente método dentro de `VistaJuego`:

```
private void salir() {
    Bundle bundle = new Bundle();
    bundle.putInt("puntuacion", puntuacion);
    Intent intent = new Intent();
    intent.putExtras(bundle);
}
```

```
padre.setResult(Activity.RESULT_OK, intent);
padre.finish();
}
```

6. Al final del método `destruyeAsteroide()` introduce:

```
if (asteroides.isEmpty()) {
    salir();
}
```

7. Al final del método `actualizaFisica()` introduce:

```
for (Grafico asteroide : asteroides) {
    if (asteroide.verificaColision(nave)) {
        salir();
    }
}
```

8. En el método `onCreate` de `Juego` introduce:

```
vistaJuego.setPadre(this);
```

9.3. Preferencias

Las preferencias (clase `SharedPreferences`) pueden usarse como un mecanismo para que los usuarios modifiquen algunos parámetros de configuración de la aplicación. Este uso se ha estudiado en el capítulo 3, donde se describe cómo podemos crear una actividad descendiente de `PreferenceFragment` para que el usuario consulte y modifique estas preferencias.

Las preferencias también pueden utilizarse como un mecanismo liviano para almacenar ciertos datos que tu aplicación quiera conservar de forma permanente. Es un mecanismo sencillo que te permite almacenar una serie de variables con su nombre y su valor. Puedes almacenar variables de tipo `boolean`, `int`, `long`, `float` y `String`. En este apartado describimos su utilización.

Las preferencias son almacenadas en ficheros XML dentro de la carpeta `shared_prefs` en los datos de la aplicación. Recuerda que en el capítulo 3 hemos visto que las preferencias de usuario siempre se almacenan en el fichero `paquete_preferencias`, donde el paquete ha de ser reemplazado por el paquete de la aplicación (en `Asteroides`, el fichero es `org.example.asteroides_preferencias`). Cuando utilices las preferencias para almacenar otros valores, podrás utilizar otros ficheros. Tienes dos alternativas según utilices uno de los siguientes métodos:

- `getSharedPreferences()`: Te permite indicar de forma explícita el nombre de un fichero de preferencias en un parámetro. Puedes utilizarlo cuando necesites varios ficheros de preferencias o acceder al mismo fichero desde varias actividades.
- `getPreferences()`: No tienes que indicar ningún nombre de fichero. Puedes utilizarlo cuando solo necesites un fichero de preferencias en la actividad.

Estos dos métodos necesitan como parámetro el tipo de acceso que queramos dar al fichero de preferencias. Los valores posibles son `MODE_PRIVATE`, `MODE_WORLD_READABLE` o `MODE_WORLD_WRITEABLE` según queramos tener acceso exclusivo a nuestra aplicación, permitir la lectura o permitir la lectura y la escritura a otras aplicaciones.

Una llamada a uno de estos dos métodos te devolverá un objeto de la clase `SharedPreferences`.

Para escribir las preferencias puedes utilizar el siguiente código:

```
SharedPreferences preferencias= getPreferences(MODE_PRIVATE);
SharedPreferences.Editor editor = preferencias.edit();
editor.putString("nombre", "Juan");
editor.putInt("edad", 35);
editor.apply();
```

Para leer las preferencias puedes utilizar el siguiente código:

```

SharedPreferences preferencias= getPreferences(MODE_PRIVATE);
String nombre = preferencias.getString("nombre",
                                       "valor por defecto");

int edad = preferencias.getInt("edad", -1);

```

El ejemplo anterior puede ser modificado reemplazando `getPreferences()` por `getSharedPreferences()`. En este caso, tendrás que indicar el fichero donde se almacenarán las preferencias.



Vídeo[tutorial]: Almacenar información usando Preferencias



Ejercicio: Almacenando la última puntuación en un fichero de preferencias

Veamos un ejemplo de cómo podemos crear un fichero de preferencias para almacenar la última puntuación obtenida en Asteroides.

1. Abre el proyecto Asteroides.
2. Crea una nueva clase `AlmacenPuntuacionesPreferencias`.
3. Reemplaza el código por el siguiente:

```

public class AlmacenPuntuacionesPreferencias implements AlmacenPuntuaciones {
    private static String PREFERENCIAS = "puntuaciones";
    private Context context;
    public AlmacenPuntuacionesPreferencias(Context context) {
        this.context = context;
    }

    public void guardarPuntuacion(int puntos, String nombre,
                                  long fecha) {
        SharedPreferences preferencias =context.getSharedPreferences(
            PREFERENCIAS, Context.MODE_PRIVATE);
        SharedPreferences.Editor editor = preferencias.edit();
        editor.putString("puntuacion", puntos + " " + nombre);
        editor.apply();
    }

    public List<String> listaPuntuaciones(int cantidad) {
        List<String> result = new ArrayList<String>();
        SharedPreferences preferencias =context.getSharedPreferences(
            PREFERENCIAS, Context.MODE_PRIVATE);
        String s = preferencias.getString("puntuacion", "");
        if (!s.isEmpty()) {
            result.add(s);
        }
        return result;
    }
}

```

4. Abre el fichero `MainActivity.java` y modifica el método `onCreate()` para que la variable `almacen` se inicialice de la siguiente manera:

```
almacen = new AlmacenPuntuacionesPreferencias(this);
```

5. Ejecuta el proyecto y verifica que la última puntuación se guarda correctamente.
6. Selecciona la pestaña `Device File Explorer` en la esquina inferior derecha. Verifica que se ha creado el fichero `/data/data/org.example.asteroides/ shared_prefs/puntuaciones.xml`.
7. Haz doble clic sobre el fichero y observa su contenido.



Práctica: Almacenando las últimas 10 puntuación en un fichero de preferencias

En el ejercicio anterior solo hemos guardado la última puntuación, lo cual no coincide con la idea planteada en un principio: nos interesaba guardar una lista con las últimas puntuaciones. En esta práctica has de tratar de solucionar este inconveniente. **NOTA:** Se trata de una práctica básicamente de programación en Java. Si no estás interesado puedes consultar directamente la solución.

1. Las preferencias solo están preparadas para almacenar variables de tipos simple, por lo que no permiten almacenar una lista. Para solucionar este inconveniente, te recomendamos que crees 10 preferencias que se llamen `puntuacion0`, `puntuacion1`, ..., `puntuacion9`.
2. Cuando se llame a `guardarPuntuacion()` almacena la nueva puntuación en `puntuacion0`. Pero antes ten la precaución de copiar el valor de `puntuacion8` en `puntuacion9`; `puntuacion7` en `puntuacion8`, y así hasta la primera. Esta operación puede realizarse por medio de un bucle con un índice entero, `n`, de forma que el nombre de la preferencia a mover puedes expresarlo como `"puntuacion"+n`.
3. Utiliza el mismo truco para implementar el método `listaPuntuaciones()`.



Solución: Almacenando las últimas 10 puntuación en un fichero de preferencias

1. Reemplaza en `guardarPuntuacion()`:

```
editor.putString("puntuacion", puntos + " " + nombre);
```

por:

```
for (int n = 9; n >= 1; n--) {
    editor.putString("puntuacion" + n,
        preferencias.getString("puntuacion" + (n - 1), ""));
}
editor.putString("puntuacion0", puntos + " " + nombre);
```

2. Reemplaza en `listaPuntuaciones()`:

```
String s = preferencias.getString("puntuacion", "");
if (!s.isEmpty()) {
    result.add(s);
}
```

por:

```
for (int n = 0; n <= 9; n++) {
    String s = preferencias.getString("puntuacion" + n, "");
    if (!s.isEmpty()) {
        result.add(s);
    }
}
```

9.4. Accediendo a ficheros

Existen tres tipos de ficheros donde podemos almacenar información en Android: ficheros almacenados en la memoria interna del teléfono, ficheros almacenados en la memoria externa (normalmente una tarjeta SD) y ficheros almacenados en los recursos. Estos últimos son de solo lectura, por lo que no son útiles para almacenar información desde la aplicación. Cuando programes en Android debes tener en cuenta que un dispositivo móvil tiene una capacidad de almacenamiento limitada.



Vídeo[tutorial]: Gestión de ficheros en Android

9.4.1. Sistema interno de ficheros

Android permite almacenar ficheros en la memoria interna del teléfono. Por defecto, los ficheros almacenados solo son accesibles para la aplicación que los creó, no pueden ser leídos por otras aplicaciones, ni siquiera por el usuario del teléfono. Cada aplicación dispone de una carpeta especial para almacenar ficheros (`/data/data/nombre_del_paquete/files`). La ventaja de utilizar esta carpeta es que cuando se desinstala la aplicación los ficheros que has creado se eliminarán. Cuando trabajes con ficheros en Android, ten siempre en cuenta que la memoria disponible de los teléfonos móviles es limitada.

Recuerda que el sistema de ficheros se sustenta en la capa Linux, por lo que Android hereda su estructura. Cuando se instala una nueva aplicación, Android crea un nuevo usuario Linux asociado a la aplicación y es este usuario el que podrá o no acceder a los ficheros.

Puedes utilizar cualquier rutina del paquete `java.io` para trabajar con ficheros. Adicionalmente se han creado métodos adicionales asociados a la clase `Context` para facilitarte el trabajo con ficheros almacenados en la memoria interna. En particular, los métodos `openFileInput()` y `openFileOutput()` te permiten abrir un fichero para lectura o escritura respectivamente. Si utilizas estos métodos, el nombre del archivo no puede contener subdirectorios. De hecho, el fichero siempre se almacena en la carpeta reservada para tu aplicación (`/data/data/nombre_del_paquete/files`). Recuerda cerrar siempre los ficheros con el método `close()`. El siguiente ejemplo muestra cómo crear un fichero y escribir en él un texto:

```
String fichero = "fichero.txt";
String texto = "texto almacenado";
FileOutputStream fos;
try {
    fos = openFileOutput(fichero, Context.MODE_PRIVATE);
    fos.write(texto.getBytes());
    fos.close();
} catch (FileNotFoundException e) {
    Log.e("Mi Aplicación", e.getMessage(), e);
} catch (IOException e) {
    Log.e("Mi Aplicación", e.getMessage(), e);
}
```

Es muy importante hacer un manejo cuidadoso de los errores. De hecho, el acceso a ficheros ha de realizarse de forma obligatoria dentro de una sección `try/catch`.

Además de los dos métodos indicados, pueden serte útiles algunos de los siguientes: `getFilesDir()` devuelve la ruta absoluta donde se están guardando los ficheros; `getDir()` crea un directorio en tu almacenamiento interno (o lo abre si existe); `deleteFile()` borra un fichero; `fileList()` devuelve un `array` con los ficheros almacenados por tu aplicación.



Ejercicio: Almacenando puntuaciones en un fichero de la memoria interna

El siguiente ejercicio muestra una clase que implementa la interfaz `AlmacenPuntuaciones` utilizando los métodos antes descritos.

1. Abre el proyecto Asteroides.
2. Crea una nueva clase `AlmacenPuntuacionesFicheroInterno`.
3. Reemplaza el código por el siguiente:

```
public class AlmacenPuntuacionesFicheroInterno implements AlmacenPuntuaciones {
    private static String FICHERO = "puntuaciones.txt";
```



```

private Context context;

public AlmacenPuntuacionesFicheroInterno(Context context) {
    this.context = context;
}

public void guardarPuntuacion(int puntos, String nombre, long fecha){
    try {
        FileOutputStream f = context.openFileOutput(FICHERO,
                                                    Context.MODE_APPEND);
        String texto = puntos + " " + nombre + "\n";
        f.write(texto.getBytes());
        f.close();
    } catch (Exception e) {
        Log.e("Asteroides", e.getMessage(), e);
    }
}

public List<String> listaPuntuaciones(int cantidad) {
    List<String> result = new ArrayList<String>();
    try {
        FileInputStream f = context.openFileInput(FICHERO);
        BufferedReader entrada = new BufferedReader(
            new InputStreamReader(f));

        int n = 0;
        String linea;
        do {
            linea = entrada.readLine();
            if (linea != null) {
                result.add(linea);
                n++;
            }
        } while (n < cantidad && linea != null);
        f.close();
    } catch (Exception e) {
        Log.e("Asteroides", e.getMessage(), e);
    }
    return result;
}
}

```

4. Abre el fichero *MainActivity.java* y en el método `onCreate()` reemplaza la línea adecuada por:

```
almacen = new AlmacenPuntuacionesFicheroInterno(this);
```



Práctica: Configurar almacenamiento de puntuaciones desde preferencias

Modifica las preferencias de la aplicación Asteroides para que el usuario pueda seleccionar dónde se guardarán las puntuaciones. De momento incluye tres opciones: "Array", "Preferencias" y "Fichero en memoria interna".

1. Abre el fichero *MainActivity.java* y en el método `onCreate()` reemplaza:

```
almacen = new AlmacenPuntuacionesFicheroInterno(this);
```

por el código necesario para que se inicialice la variable `almacen` de forma adecuada según el valor introducido en preferencias.

2. Verifica el resultado.
3. Observa que cuando desde las preferencias cambias de tipo de almacenamiento, no tiene efecto hasta que sales de la actividad principal y cargas de nuevo la aplicación. Para resolverlo, arranca la actividad de preferencias usando `startActivityForResult()`. En el método `onActivityResult()` verifica si se vuelve de esta actividad e inicializa de nuevo la variable `almacen`.

4. Verifica de nuevo el resultado.
5. Cada vez que añadas un nuevo método de almacenamiento inclúyelo en la lista de preferencias.



Preguntas de repaso: Ficheros

9.4.2. Sistema de almacenamiento externo

Los teléfonos Android suelen disponer de memoria adicional de almacenamiento, conocido como almacenamiento externo. Este almacenamiento suele ser de mayor capacidad, por lo que resulta ideal para almacenar ficheros de música o vídeo. Suele ser una memoria extraíble, como una tarjeta SD, o una memoria interna no extraíble (algunos modelos incorporan los dos tipos de memoria, es decir, almacenamiento externo extraíble y almacenamiento interno no extraíble). Cuando conectamos el dispositivo Android a través del cable USB permitimos el acceso a esta memoria externa, de forma que los ficheros aquí escritos podrán ser leídos, modificados o borrados por cualquier usuario.

Para acceder a la memoria externa, lo habitual es utilizar la ruta `/sdcard/...`

Esta es la carpeta donde el sistema monta la tarjeta SD. No obstante, resulta más conveniente utilizar el método `Environment.getExternalStorageDirectory()` para que el sistema nos indique la ruta exacta.

Resulta necesario solicitar el permiso `WRITE_EXTERNAL_STORAGE` en `AndroidManifest.xml` para poder escribir en la memoria externa. En la versión 4.1 aparece el permiso `READ_EXTERNAL_STORAGE`. Sin embargo, este permiso se ha introducido para un futuro uso. En la actualidad todas las aplicaciones pueden leer en la memoria externa. Por lo tanto, has de tener cuidado con la información que dejas en esta memoria.



Vídeo[tutorial]: Almacenamiento externo en Android



Ejercicio: Almacenando puntuaciones en la memoria externa

1. Abre el proyecto del ejercicio anterior.
2. Selecciona el fichero `AlmacenPuntuacionesFicheroInterno.java` y cópialo en el portapapeles (`Ctrl-C`).
3. Pega el fichero sobre el proyecto (`Ctrl-V`) y renómbralo como `AlmacenPuntuacionesFicheroExterno.java`.
4. Abre la nueva clase creada y reemplaza la inicialización de la variable `FICHERO` por:

```
private static String FICHERO = Environment.  
    getExternalStorageDirectory() + "/puntuaciones.txt";
```

Dependiendo de si utilizas un emulador o un dispositivo real, el valor de `FICHERO` será diferente. Posibles valores son: `"/sdcard/puntuaciones.txt"` o `"/storage/sdcard0/puntuaciones.txt"`.

5. En el método `guardarPuntuacion()` reemplaza la inicialización de `f` por:

```
FileOutputStream f = new FileOutputStream(FICHERO, true);
```

6. En el método `listaPuntuaciones()` reemplaza la inicialización de `f` por:

```
FileInputStream f = new FileInputStream(FICHERO);
```

- En el método `onCreate()` de la actividad `MainActivity` reemplaza la inicialización de `almacen` por:

```
almacen = new AlmacenPuntuacionesFicheroExterno(this);
```

O si has hecho la práctica *Configurar almacenamiento de puntuaciones desde preferencias* añade un nuevo tipo en las preferencias.

- Abre el fichero `AndroidManifest.xml` y solicita el permiso `WRITE_EXTERNAL_STORAGE`. No es imprescindible que pidas permiso de lectura, dado que este ya está implícito cuando se pide el de escritura. Se trata de un permiso peligroso. Si ejecutas la aplicación en un dispositivo con versión 6 o superior, debes dar el permiso de forma manual desde los ajustes del terminal.
- Ejecuta la aplicación y crea nuevas puntuaciones.
- Verifica con la vista *File Explorer* que en la carpeta `sdcard` aparece el fichero.



Práctica: Solicitar permiso de acceso a memoria externa

Antes de leer o escribir el fichero en la memoria externa, comprueba que tienes permiso. En caso negativo, solicítalo al usuario. Puedes basarte en el ejercicio: *Solicitud de permisos en Android Marshmallow*.

Verificando acceso a la memoria externa

La memoria externa puede haber sido extraída o estar protegida contra escritura. Puedes utilizar el método `Environment.getExternalStorageState()` para verificar el estado de la memoria. Veamos cómo se utiliza:

```
String stadoSD = Environment.getExternalStorageState();
if (stadoSD.equals(Environment.MEDIA_MOUNTED)) {
    // Podemos leer y escribir
    ...
} else if (stadoSD.equals(Environment.MEDIA_MOUNTED_READ_ONLY)) {
    // Podemos leer
    ...
} else {
    // No podemos leer y ni escribir
    ...
}
```



Práctica: Verificando acceso a la memoria externa

- Modifica la clase `AlmacenPuntuacionesFicheroExterno` para que antes de acceder a la memoria externa verifique que la operación es posible. En caso contrario mostrará un `Toast` y saldrá del método.
- Ejecuta el programa en un dispositivo real con memoria externa y verifica que se almacena correctamente.
- Ahora verifica el comportamiento cuando la memoria externa no está disponible. Para que el dispositivo ya no tenga acceso a esta memoria, la solución más sencilla consiste en conectar el dispositivo con el cable USB y activar el almacenamiento por USB.



Solución: Verificando acceso a la memoria externa

- En `guardarPuntuacion()` añade:

```
String stadoSD = Environment.getExternalStorageState();
if (!stadoSD.equals(Environment.MEDIA_MOUNTED)) {
    Toast.makeText(context, "No puedo escribir en la memoria externa",
        Toast.LENGTH_LONG).show();

    return;
}
```

2. En `listaPuntuaciones()` añade:

```
String stadoSD = Environment.getExternalStorageState();
if (!stadoSD.equals(Environment.MEDIA_MOUNTED) &&
    !stadoSD.equals(Environment.MEDIA_MOUNTED_READ_ONLY)) {
    Toast.makeText(context, "No puedo leer en la memoria externa",
        Toast.LENGTH_LONG).show();

    return result;
}
```

Almacenando ficheros específicos de tu aplicación en el almacenamiento externo

Las aplicaciones pueden almacenar los ficheros en una carpeta específica del sistema de almacenamiento externo, de forma que cuando la aplicación sea desinstalada se borren automáticamente estos ficheros. En concreto, esta carpeta ha de seguir esta estructura:

```
/Android/data/<nombre_del_paquete>/files/
```

Donde el paquete `<nombre_del_paquete>` ha de cambiarse por el nombre del paquete de la aplicación, por ejemplo `org.example.asteroides`.

Una gran ventaja de trabajar en este almacenamiento es que a partir del API 19 (v4.4) no es necesario pedir permiso de almacenamiento.

Puedes utilizar el método `getExternalFilesDir(null)` para obtener esta ruta. Si en lugar de `null` indicas alguna de las constantes que se indican más abajo, se devolverá la ruta a una carpeta específica según el tipo de contenido que nos interese. Este método crea la carpeta en caso de no existir previamente. Indicando la carpeta garantizamos que el escáner de medios de Android categoriza los ficheros de forma adecuada. Por ejemplo, un tono de llamada será identificado como tal y no como un fichero de música. De esta forma, no aparecerá en la lista de música que puede reproducir el reproductor multimedia. Estas carpetas también son eliminadas cuando se desinstala la aplicación.

Constante	Carpeta	Descripción
<code>DIRECTORY_MUSIC</code>	<code>Music</code>	Ficheros de música
<code>DIRECTORY_PODCASTS</code>	<code>Podcasts</code>	Descargas desde podcast
<code>DIRECTORY_RINGTONES</code>	<code>Ringtones</code>	Tono de llamada de teléfono
<code>DIRECTORY_ALARMS</code>	<code>Alarms</code>	Sonidos de alarma
<code>DIRECTORY_NOTIFICATIONS</code>	<code>Notifications</code>	Sonidos para notificaciones
<code>DIRECTORY_PICTURES</code>	<code>Pictures</code>	Ficheros con fotografías
<code>DIRECTORY_DOWNLOADS</code>	<code>Download</code>	Descargas de cualquier tipo
<code>DIRECTORY_DCIM</code>	<code>DCIM</code>	Carpeta que tradicional-mente crean las cámaras



Práctica: Almacenando puntuaciones en una carpeta de la aplicación de la memoria externa

1. Selecciona el fichero `AlmacenPuntuacionesFicheroExterno.java`, y cópialo en el portapapeles (`Ctrl-C`).
2. Pega el fichero sobre el proyecto (`Ctrl-V`) y renómbralo como `AlmacenPuntuacionesFicheroExtApl.java`.

3. Modifica los métodos `listaPuntuaciones()` y `guardarPuntuacion()` para que las puntuaciones se almacenen en la memoria externa, pero en una carpeta de tu aplicación.
4. Modifica el código correspondiente para que la nueva clase pueda ser seleccionada como almacén de las puntuaciones.
5. Ejecuta la aplicación. Desinstala la aplicación y verifica si el fichero ha sido eliminado.

Almacenando ficheros compartidos en el almacenamiento externo

Si quieres crear un fichero que no sea específico para tu aplicación y quieres que no sea borrado cuando tu aplicación sea desinstalada, puedes crearlo en cualquier otro directorio del almacenamiento externo.

Lo ideal es que utilices alguno de los directorios públicos creados para almacenar diferentes tipos de ficheros. Estos directorios parten de la raíz del almacenamiento externo y siguen con alguna de las carpetas listadas en la tabla anterior.

A partir del nivel de API 8 puedes utilizar el método `getExternalStoragePublicDirectory(String tipo)` para obtener esta ruta de uno de estos directorios compartidos. Como parámetro utiliza alguna de las constantes que se indican en la tabla anterior. Guardando los ficheros en las carpetas adecuadas garantizamos que el escáner de medios de Android categoriza los ficheros de forma adecuada. Si utilizas un nivel de API anterior al 8, lo recomendable es crear estas carpetas manualmente.

NOTA: Si quieres que tus ficheros estén ocultos al escáner de medios, incluye un fichero vacío que se llame `.nomedia` en la carpeta donde estén almacenados.

Almacenando externo con varias unidades

Algunos dispositivos incluyen varias unidades de almacenamiento externo. En este caso, al conectar el dispositivo con un cable USB a un ordenador aparecerá más de una unidad:



En estos casos, una unidad suele corresponder a una tarjeta extraíble SD y la otra una partición en la memoria *flash*. Si utilizamos el método `getExternalFilesDir()`, y los relacionamos, nos devolverá una de las unidades. Esta unidad se denomina unidad de almacenamiento primaria y el resto de unidades, secundarias. Es el fabricante quien decide cuál de las unidades es la memoria primaria. Normalmente Samsung escoge como memoria externa primaria la partición *flash* no extraíble.

Hasta la versión 4.4 el API de Android no soportaba múltiples unidades de memoria externa. Solo podíamos acceder de forma estándar a la memoria externa primaria y para acceder a la memoria externa secundaria es necesario conocer dónde el fabricante ha montado esta memoria. En la mayoría de los casos se monta en `/mnt/sdcard/external_sd`.

A partir de la versión 4.4 se incorporan varios métodos que nos permiten trabajar con varias unidades externas. En la clase `Context` se añade `File[] getExternalFilesDirs(String)`, que nos devuelve un array con la ruta a cada uno de los almacenamientos externos disponibles. El primer elemento ha de coincidir con la ruta devuelta por `getExternalFilesDir(String)`. La clase `Environment` incorpora el método estático `String getStorageState(File)`, que permite conocer el estado de cada unidad de almacenamiento. Nos devuelve una información equivalente a la del método `getExternalStorageState()`.



Desafío: Permitir seleccionar diferentes almacenamientos externos

En caso de existir varios almacenamientos externos, se mostrará al usuario un listado para que seleccione dónde se almacenarán las puntuaciones. Solo podrá realizarse en versiones superiores a la 4.4.



Preguntas de repaso: La memoria externa

9.4.3. Acceder a un fichero de los recursos

También tienes la posibilidad de almacenar ficheros en los recursos, es decir, adjuntos al paquete de la aplicación. Has de tener en cuenta que estos ficheros no podrán ser modificados.

Tienes dos alternativas para esto: usar la carpeta `res/raw` o `assets`. La principal diferencia a la hora de usar una carpeta u otra está en la forma de identificar el fichero. Por ejemplo, si arrastras un fichero que se llame `datos.txt` a la carpeta `res/raw`, podrás acceder a él usando `context.getResources().openRawResource(R.raw.datos)`. Si, por el contrario, dejas este fichero en la carpeta `assets`, podrás acceder a él usando `context.getAssets().open("datos.txt")`. Otra diferencia es que dentro de `assets` podrás crear subcarpetas para organizar los ficheros.

Recuerda que, tanto en la carpeta `raw` como en `assets`, los ficheros nunca son comprimidos.



Ejercicio: Leyendo puntuaciones de un fichero de recursos en `res/raw`

1. Con el explorador de ficheros busca en el terminal un fichero de texto que se llame `puntuaciones.txt`, creado en alguno de los ejercicios anteriores.
2. Extráelo del terminal y pégalo en la carpeta `res/raw` del proyecto Asteroides.
3. Selecciona el fichero `AlmacenPuntuacionesFicheroInterno.java` y cópialo en el portapapeles (`Ctrl-C`).
4. Pega el fichero sobre el proyecto (`Ctrl-V`) y renómbralo como `AlmacenPuntuacionesRecursoRaw.java`.
5. Elimina de esta clase todo el código del método `guardarPuntuacion()`. No se realiza ninguna acción en este método.
6. Para que las puntuaciones se lean del fichero de los recursos, en el método `listaPuntuaciones()` reemplaza:

```
FileInputStream f = context.openFileInput(FICHERO);
```

por:

```
InputStream f = context.getResources().openRawResource(
    R.raw.puntuaciones);
```

7. La siguiente línea ya no tiene sentido. Elimínala:

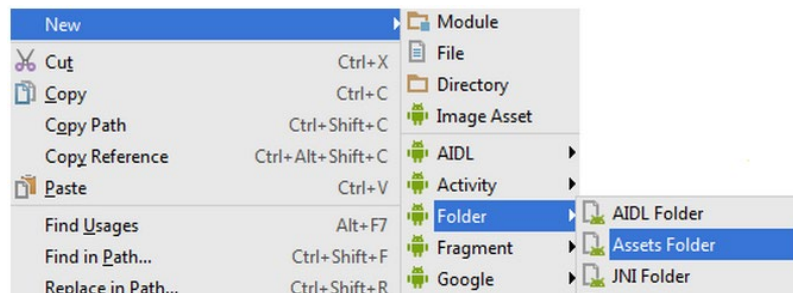
```
private static String FICHERO = "puntuaciones.txt";
```

8. Modifica el código correspondiente para que la nueva clase pueda ser seleccionada como almacén de las puntuaciones.
9. Verifica el resultado.

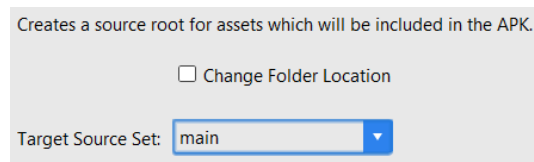


Ejercicio: Leyendo puntuaciones de un fichero de recursos en `assets`

1. Selecciona `File / New / Folder / Assets Folder`:



En la siguiente ventana deja los valores por defecto:



Hemos creado la carpeta *assets* que aparecerá dentro de *res*. Vamos a crear la subcarpeta *carpeta* dentro de esta carpeta. Pulsa con el botón derecho sobre *assets*, selecciona *New/Directory* e introduce "carpeta".

2. Copia el fichero *puntuaciones.txt* dentro de la carpeta que acabas de crear.
3. Selecciona el fichero *AlmacenPuntuacionesRecursoRaw.java* y cópialo en el portapapeles (*Ctrl-C*).
4. Pega el fichero sobre el proyecto (*Ctrl-V*) y renómbralo como *AlmacenPuntuacionesRecursoAssets.java*.
5. En el método *listaPuntuaciones()* reemplaza:

```
InputStream f = context.getResources().openRawResource(
    R.raw.puntuaciones);
```

por:

```
InputStream f = context.getAssets().open("carpeta/puntuaciones.txt");
```

6. Modifica el código correspondiente para que la nueva clase pueda ser seleccionada como almacén de las puntuaciones.
7. Verifica que el resultado es idéntico al ejercicio anterior.

9.5. Trabajando con XML

Como sabrás, XML es uno de los estándares más utilizados en la actualidad para codificar información. Es ampliamente utilizado en Internet; además, como hemos mostrado a lo largo de este libro, se utiliza para múltiples usos en el SDK de Android. Entre otras cosas, es utilizado para definir *layouts*, animaciones, *AndroidManifest.xml*, etc.

Una de las mayores fortalezas de la plataforma Android es que se aprovecha el lenguaje de programación Java y sus librerías. El SDK de Android no acaba de ofrecer todo lo disponible para su estándar del entorno de ejecución Java (JRE), pero es compatible con una fracción muy significativa de este. Lo mismo ocurre en lo referente a trabajar con XML: Java dispone de una gran cantidad de API con este propósito, pero no todas están disponibles desde Android.

Librerías disponibles:

Java's Simple API for XML (SAX) (paquetes `org.xml.sax.*`).

Document Object Model (DOM) (paquetes `org.w3c.dom.*`).

Librerías no disponibles:

Streaming API for XML (StAX). Aunque se dispone de otra librería con funcionalidad equivalente (paquete `org.xmlpull.v1.XmlPullParser`).

Java Architecture for XML Binding (JAXB). Resultaría demasiado pesada para Android.

Como podrás ver al estudiar los ejemplos, leer y escribir ficheros XML es muy laborioso y necesitarás algo de esfuerzo para comprender el código empleado. Vamos a explicar las dos alternativas más importantes, SAX y DOM. El planteamiento es bastante diferente. Tras ver los ejemplos podrás decidir qué herramienta se adapta mejor a tus gustos personales o al problema en concreto que tengas que resolver.

El ejemplo utilizado para ilustrar el trabajo con XML será el mismo que el utilizado en el resto del capítulo: almacenar las mejores puntuaciones obtenidas. El formato XML que se utilizará para este propósito se muestra a continuación:

```
<?xml version="1.0" encoding="UTF-8"?>
<lista_puntuaciones>
  <puntuacion fecha="1288122023410">
    <nombre>Mi nombre</nombre>
    <puntos>45000</puntos>
  </puntuacion>
  <puntuacion fecha="1288122428132">
    <nombre>Otro nombre</nombre>
    <puntos>31000</puntos>
  </puntuacion>
</lista_puntuaciones>
```

9.5.1. Procesando XML con SAX

El uso de la API SAX (*Simple API for XML*) se recomienda cuando se desea un programa de análisis rápido y se quiere reducir al mínimo el consumo de memoria de la aplicación. Eso hace que sea muy apropiada para un dispositivo móvil con Android. También resulta ventajosa para procesar ficheros de gran tamaño.

SAX nos facilita realizar un *parser* (analizador) sobre un documento XML para así poder analizar su estructura. Ha de quedar claro que SAX no almacena los datos. Por lo tanto, necesitaremos una estructura de datos donde guardar la información contenida en el XML. Para realizar este *parser* se generarán una serie de eventos a medida que se vaya leyendo el documento secuencialmente. Por ejemplo, al analizar el documento XML anterior, SAX generará los siguientes eventos:

```
Comienza elemento: lista_puntuaciones
Comienza elemento: puntuacion, con atributo fecha="1288122023410"
Comienza elemento: nombre
Texto de nodo: Mi nombre
Finaliza elemento: nombre
Comienza elemento: puntos
Texto de nodo: 45000
Finaliza elemento: puntos
Finaliza elemento: puntuacion
Comienza elemento: puntuacion, con atributo fecha="1288122428132"
Comienza elemento: nombre
Texto de nodo: Otro nombre
Finaliza elemento: nombre
Comienza elemento: puntos
Texto de nodo: 31000
Finaliza elemento: puntos
Finaliza elemento: puntuacion
Finaliza elemento: lista_puntuaciones
```

Para analizar un documento mediante SAX, vamos a escribir métodos asociados a cada tipo de evento. Este proceso se realiza extendiendo la clase `DefaultHandler`, que nos permite reescribir 5 métodos. Los métodos listados a continuación serán llamados a medida que ocurran los eventos listados anteriormente.

`startDocument()`: Comienza el documento XML.

`endDocument()`: Finaliza documento XML.

startElement(String uri, String nombreLocal, String nombreCualif, Attributes atributos): Comienza una nueva etiqueta; se indican los parámetros:

uri: La uri del espacio de nombres o vacío, si no se ha definido.

nombreLocal: Nombre local de la etiqueta sin prefijo.

nombreCualif: Nombre cualificado de la etiqueta con prefijo.

atributos: Lista de atributos de la etiqueta.

endElement(String uri, String nombreLocal, String nombreCualif): Termina una etiqueta.

characters(char ch[], int comienzo, int longitud): Devuelve en **ch** los caracteres dentro de una etiqueta. Es decir, en `<etiqueta> caracteres </etiqueta>` devolvería **caracteres**. Para obtener un **String** con estos caracteres: `String s = new String(ch,comienzo,longitud)`. Más adelante veremos un ejemplo de cómo utilizar este método.



Ejercicio: Almacenando puntuaciones en XML con SAX

Una vez descritos los principios de trabajo con SAX, pasemos a implementar la interfaz **AlmacenPuntuaciones** mediante esta API.

1. Crea la clase **AlmacenPuntuacionesXML_SAX** en la aplicación **Asteroides** y escribe el siguiente código:

```
public class AlmacenPuntuacionesXML_SAX implements AlmacenPuntuaciones {
    private static String FICHERO = "puntuaciones.xml";
    private Context contexto;
    private ListaPuntuaciones lista;
    private boolean cargadaLista;

    public AlmacenPuntuacionesXML_SAX(Context contexto) {
        this.contexto = contexto;
        lista = new ListaPuntuaciones();
        cargadaLista = false;
    }

    @Override
    public void guardarPuntuacion(int puntos, String nombre, long fecha) {
        try {
            if (!cargadaLista){
                lista.leerXML(contexto.openFileInput(FICHERO));
            }
        } catch (FileNotFoundException e) {
        } catch (Exception e) {
            Log.e("Asteroides", e.getMessage(), e);
        }
        lista.nuevo(puntos, nombre, fecha);
        try {
            lista.escribirXML(contexto.openFileOutput(FICHERO,
                Context.MODE_PRIVATE));
        } catch (Exception e) {
            Log.e("Asteroides", e.getMessage(), e);
        }
    }

    @Override
    public List<String> listaPuntuaciones(int cantidad) {
        try {
            if (!cargadaLista){
                lista.leerXML(contexto.openFileInput(FICHERO));
            }
        } catch (Exception e) {
            Log.e("Asteroides", e.getMessage(), e);
        }
        return lista.aListString();
    }
}
```

}

La nueva clase comienza definiendo una serie de variables y constantes. En primer lugar, el nombre del fichero donde se guardarán los datos. Con el valor indicado, el fichero se almacenará en `/data/data/org.example.asteroides/files/puntuaciones.xml`. Pero puedes almacenarlos en otro lugar, como por ejemplo en la memoria SD. La variable más importante es `lista` de la clase `ListaPuntuaciones`. En ella guardaremos la información contenida en el fichero XML. Esta clase se define a continuación. La variable `cargadaLista` nos indica si `lista` ya ha sido leída desde el fichero.

El código continúa sobrescribiendo los dos métodos de la interfaz. En `guardarPuntuacion()` comenzamos verificando si `lista` ya ha sido cargada, para hacerlo en caso necesario. Es posible que el programa se esté ejecutando por primera vez, en cuyo caso el fichero no existirá. En este caso se producirá una excepción de tipo `FileNotFoundException` al tratar de abrir el fichero. Esta excepción es capturada por nuestro código, pero no realizamos ninguna acción dado que no se trata de un verdadero error. A continuación se añade un nuevo elemento a `lista` y se escribe de nuevo el fichero XML. El siguiente método, `listaPuntuacion()`, resulta sencillo de entender, al limitarse a métodos definidos en la clase `ListaPuntuaciones`.

2. Pasemos a mostrar el comienzo de la clase `ListaPuntuaciones`. No es necesario almacenarla en un fichero aparte, puedes definirla dentro de la clase anterior. Para ello copia el siguiente código justo antes del último `}` de la clase `AlmacenPuntuacionesXML_SAX`:

```
private class ListaPuntuaciones {
    private class Puntuacion {
        int puntos;
        String nombre;
        long fecha;
    }

    private List<Puntuacion> listaPuntuaciones;

    public ListaPuntuaciones() {
        listaPuntuaciones = new ArrayList<Puntuacion>();
    }

    public void nuevo(int puntos, String nombre, long fecha) {
        Puntuacion puntuacion = new Puntuacion();
        puntuacion.puntos = puntos;
        puntuacion.nombre = nombre;
        puntuacion.fecha = fecha;
        listaPuntuaciones.add(puntuacion);
    }

    public List<String> aListString() {
        List<String> result = new ArrayList<String>();
        for (Puntuacion puntuacion : listaPuntuaciones) {
            result.add(puntuacion.nombre+" "+puntuacion.puntos);
        }
        return result;
    }
}
```

El objetivo de esta clase es mantener una lista de objetos `Puntuacion`. Dispone de métodos para insertar un nuevo elemento (`nuevo()`) y devolver una lista con todas las puntuaciones almacenadas (`aListString()`).

3. Lo verdaderamente interesante de esta clase es que permite la lectura y escritura de los datos desde un documento XML (`leerXML()` y `escribirXML()`). Veamos primero cómo leer un documento XML usando SAX. Escribe el siguiente código a continuación del anterior:

```
public void leerXML(InputStream entrada) throws Exception {
    SAXParserFactory fabrica = SAXParserFactory.newInstance();
    SAXParser parser = fabrica.newSAXParser();
    XMLReader lector = parser.getXMLReader();
    ManejadorXML manejadorXML = new ManejadorXML();
```

```
lector.setContentHandler(manejadorXML);
lector.parse(new InputSource(entrada));
cargadalista = true;
}
```

Para leer un documento XML comenzamos creando una instancia de la clase `SAXParserFactory`, lo que nos permite crear un nuevo *parser* XML de tipo `SAXParser`. Luego creamos un lector, de la clase `XMLReader`, asociado a este *parser*. Creamos `manejadorXML` de la clase `ManejadorXML` y asociamos este manejador al `XMLReader`. Para finalizar, le indicamos al `XMLReader` qué entrada tiene para que realice el proceso de *parser*. Una vez finalizado el proceso, marcamos que el fichero está cargado.

Como ves, el proceso es algo largo, pero siempre se realiza igual. Donde sí que tendremos que trabajar algo más es en la creación de la clase `ManejadorXML`, dado que va a depender del formato del fichero que queramos leer. Esta clase se lista en el siguiente punto.

4. Escribe este código a continuación del anterior:

```
class ManejadorXML extends DefaultHandler {
    private StringBuilder cadena;
    private Puntuacion puntuacion;

    @Override
    public void startDocument() throws SAXException {
        listaPuntuaciones = new ArrayList<Puntuacion>();
        cadena = new StringBuilder();
    }

    @Override
    public void startElement(String uri, String nombreLocal, String
        nombreCualif, Attributes atr) throws SAXException {
        cadena.setLength(0);
        if (nombreLocal.equals("puntuacion")) {
            puntuacion = new Puntuacion();
            puntuacion.fecha = Long.parseLong(atr.getValue("fecha"));
        }
    }

    @Override
    public void characters(char ch[], int comienzo, int lon) {
        cadena.append(ch, comienzo, lon);
    }

    @Override
    public void endElement(String uri, String nombreLocal,
        String nombreCualif) throws SAXException {
        if (nombreLocal.equals("puntos")) {
            puntuacion.puntos = Integer.parseInt(cadena.toString());
        } else if (nombreLocal.equals("nombre")) {
            puntuacion.nombre = cadena.toString();
        } else if (nombreLocal.equals("puntuacion")) {
            listaPuntuaciones.add(puntuacion);
        }
    }

    @Override
    public void endDocument() throws SAXException {}
}
```

Esta clase define un manejador que captura los cinco eventos generados en el proceso de *parsing* en SAX. En `startDocument()` nos limitamos a inicializar variables. En `startElement()` verificamos que hemos llegado a una etiqueta `<puntuación>`. En tal caso, creamos un nuevo objeto de la clase `Puntuacion` e inicializamos el campo `fecha` con el valor indicado en uno de los atributos.

El método `characters()` se llama cuando aparece texto dentro de una etiqueta (`<etiqueta> caracteres </etiqueta>`). Nos limitamos a almacenar este texto en la variable `cadena` para utilizarlo en el siguiente método. SAX no nos garantiza que nos pasará todo el texto en un

solo evento: si el texto es muy extenso, se realizarán varias llamadas a este método. Por esta razón, el texto se va acumulando en `cadena`.

El método `endElement()` resulta más complejo, dado que en función de que etiqueta esté acabando realizaremos una tarea diferente. Si se trata de `</puntos>` o de `</nombre>` utilizaremos el valor de la variable `cadena` para actualizar el valor correspondiente. Si se trata de `</puntuacion>` añadimos el objeto `puntuacion` a la lista.

5. Introduce a continuación el último método de la clase `ListaPuntuaciones`, que nos permite escribir el documento XML:

```
public void escribirXML(OutputStream salida) {
    XmlSerializer serializador = Xml.newSerializer();
    try {
        serializador.setOutput(salida, "UTF-8");
        serializador.startDocument("UTF-8", true);
        serializador.startTag("", "lista_puntuaciones");
        for (Puntuacion puntuacion : listaPuntuaciones) {
            serializador.startTag("", "puntuacion");
            serializador.attribute("", "fecha",
                String.valueOf(puntuacion.fecha));
            serializador.startTag("", "nombre");
            serializador.text(puntuacion.nombre);
            serializador.endTag("", "nombre");
            serializador.startTag("", "puntos");
            serializador.text(String.valueOf(puntuacion.puntos));
            serializador.endTag("", "puntos");
            serializador.endTag("", "puntuacion");
        }
        serializador.endTag("", "lista_puntuaciones");
        serializador.endDocument();
    } catch (Exception e) {
        Log.e("Asteroides", e.getMessage(), e);
    }
}
} //Cerramos ListaPuntuaciones
} //Cerramos AlmacenPuntuacionesXML_SAX
```

Como puedes ver, todo el trabajo se realiza por medio de un objeto de la clase `XmlSerializer`, que escribe el código XML en el `OutputStream` que hemos pasado como parámetros.

6. La variable `almacen` ha de inicializarse de forma adecuada.
7. Modifica el código correspondiente para que este método pueda ser seleccionado para almacenar las puntuaciones.
8. Verifica el resultado.

9.5.2. Procesando XML con DOM

DOM (*Document Object Model*) es una API creada por W3C (World Wide Web Consortium) que nos permite manipular dinámicamente documentos XML y HTML. Android soporta el nivel de especificación 3, por lo que permite trabajar con definición de tipo de documento (DTD) y validación de documentos.

Como ya hemos comentado, el planteamiento de DOM es muy diferente del de SAX. SAX recorre todo el documento XML secuencialmente y lo analiza, pero sin almacenarlo. Por el contrario, DOM permite cargar el documento XML en memoria RAM y manipularlo directamente en memoria. DOM representa el documento como un árbol. Podremos crear nuevos nodos, borrar o modificar los existentes. Una vez dispongamos de la nueva versión, podremos almacenarlo en un fichero o mandarlo por Internet.

Trabajar con DOM tiene sus ventajas frente a SAX: por ejemplo, nos evitamos definir a mano el proceso de *parser* y crear una estructura para almacenar los datos. Pero también tiene sus inconvenientes: recorrer un documento DOM puede ser algo complejo; además, al tener que cargarse todo el documento en memoria puede consumir excesivos recursos para un dispositivo

como un teléfono móvil. Este inconveniente cobra especial relevancia al trabajar con documentos grandes. Para terminar, DOM procesa la información de forma más lenta.



Enlaces de interés: Procesado XML con DOM

<http://www.androidcurso.com/index.php/818>



Preguntas de repaso: *Trabajando con XML*

9.6. Trabajando con JSON

JSON corresponde al acrónimo de *JavaScript Object Notation*. Es un formato para representar información similar a XML, pero presenta dos ventajas frente a este: es más compacto, pues necesita menos bytes para codificar la información y el código necesario para realizar un parser es mucho menor. Estas ventajas hacen que cada vez sea más popular, especialmente en el intercambio de datos a través de la red. A continuación, se muestra cómo se codificaría el ejemplo que estamos desarrollando en este capítulo:

```
{
  "puntuaciones": [
    { "fecha": 1288122023410, "nombre": "Mi nombre", "puntos": 45000 },
    { "fecha": 1288122428132, "nombre": "Otro nombre", "puntos": 31000 }
  ]
}
```

Comparando el número de caracteres empleados frente al que se utilizó para codificar esta información en XML, podemos observar una reducción cercana al 50 %.

La plataforma Android incorpora la librería estándar `org.json` con la que podremos procesar ficheros JSON. Otra alternativa es la librería `com.google.gson` que va a resultar más sencilla de utilizar. Veamos estas dos alternativas.

9.6.1. Procesando JSON con la librería Gson

GSON es una librería de código abierto creada por Google que permite serializar objetos Java para convertirlos en un `String`. Su uso más frecuente es para convertir un objeto en su representación JSON y a la inversa.

La gran ventaja de esta librería es que puede ser usada sobre objetos de cualquier tipo de clases, incluso clases preexistentes que no has creado. Esto es posible al no ser necesario introducir código en las clases para que sean serializadas.

El código necesario es muy reducido, como se muestra a continuación:

```
private ArrayList<Puntuacion> puntuaciones=new ArrayList<>();
private Gson gson = new Gson();
private Type type = new TypeToken<List<Puntuacion>>() {}.getType();

// Convertimos la colección de datos a un String JSON
String string = gson.toJson(puntuaciones, type);

// Convertimos un String JSON a una La colección de datos
puntuaciones = gson.fromJson(string, type);
```

En este código, `puntuaciones` contiene una colección (List) de elementos de tipo `Puntuacion`. Para representar estos datos en JSON vamos a necesitar un objeto `Gson` y otro `Type`. Este último representa el tipo de datos con el que trabajamos. En la variable `string` se almacenará el

contenido de `puntuaciones` en representación JSON. En la siguiente línea se hace el proceso inverso.

La librería no solo permite transformar los datos en JSON, también podemos personalizar la serialización de los datos según las necesidades del programador. También permite excluir algunos atributos para que sean incluidos en la representación JSON.



Ejercicio: Guardar puntuaciones en JSON con la librería Gson

1. Crea la clase `Puntuacion` con el siguiente código:

```
public class Puntuacion {
    private int puntos;
    private String nombre;
    private long fecha;

    public Puntuacion(int puntos, String nombre, long fecha) {
        this.puntos = puntos;
        this.nombre = nombre;
        this.fecha = fecha;
    }
}
```

2. Sitúate al final de la clase y selecciona *Code > Generate > Getter and Setter*. Selecciona todos los atributos y pulsa OK.
3. Añade al fichero *Gradle Scripts/Bulid.gradle (Module:app)* la dependencia:

```
dependencies {
    ...
    implementation 'com.google.code.gson:gson:2.8.5'
}
```

4. Crea la clase `AlmacenPuntuacionesGson` con el siguiente código:

```
public class AlmacenPuntuacionesGson implements AlmacenPuntuaciones {
    private String string; //Almacena puntuaciones en formato JSON
    private Gson gson = new Gson();
    private Type type = new TypeToken<List<Puntuacion>>() {}.getType();

    public AlmacenPuntuacionesGson() {
        guardarPuntuacion(45000, "Mi nombre", System.currentTimeMillis());
        guardarPuntuacion(31000, "Otro nombre", System.currentTimeMillis());
    }

    @Override
    public void guardarPuntuacion(int puntos, String nombre, long fecha) {
        //string = LeerString();
        ArrayList<Puntuacion> puntuaciones;
        if (string == null) {
            puntuaciones = new ArrayList<>();
        } else {
            puntuaciones = gson.fromJson(string, type);
        }
        puntuaciones.add(new Puntuacion(puntos, nombre, fecha));
        string = gson.toJson(puntuaciones, type);
        //guardarString(string);
    }

    @Override
    public List<String> listaPuntuaciones(int cantidad) {
        //string = LeerString();
        ArrayList<Puntuacion> puntuaciones;
        if (string == null) {
            puntuaciones = new ArrayList<>();
        } else {

```

```

        puntuaciones = gson.fromJson(string, type);
    }
    List<String> salida = new ArrayList<>();
    for (Puntuacion puntuacion : puntuaciones) {
        salida.add(puntuacion.getPuntos()+" "+puntuacion.getNombre());
    }
    return salida;
}
}

```

En la variable `string` se almacenará la lista de puntuaciones en representación JSON. Para que los datos se almacenen de forma no volátil tendrías que implementar los métodos `guardarString()` y `leerString()`. La forma más sencilla sería almacenarlo en un fichero de preferencias. Otra alternativa sería guardarlo en un fichero en la memoria interna o externa. En el próximo capítulo veremos cómo mandar este `String` a través de Internet.

5. Modifica el código correspondiente para que se pueda seleccionar esta clase para el almacenamiento.
6. Ejecuta el proyecto y verifica su funcionamiento. Si visualizas el valor de `string` este debe ser:

```

[{"fecha":1478552190154,"nombre":"Mi nombre", "puntos":45000},
 {"fecha":1478552205944,"nombre":"Otro nombre", "puntos":31000}]

```

NOTA: Observa como los atributos son almacenados por orden alfabético.



Práctica: Guardar el string JSON en un fichero.

1. Implementa los métodos `guardarString(String)` y `leerString()` para que la información se almacene en un fichero de preferencias o en la memoria del dispositivo.
2. En la versión anterior, la variable `string` hacía el papel de almacén de la información. En la nueva versión, este papel ha pasado a un fichero o a una preferencia. Elimina la variable global `string` y conviértela en variable local en los métodos donde sea necesario.



Ejercicio: Guardar una clase en JSON con la librería Gson.

El resultado del ejercicio anterior es muy similar al ejemplo JSON, mostrado al principio de este apartado. Sin embargo, no es exactamente igual. En el ejemplo se muestra un objeto JSON que incluye una única propiedad con nombre "puntuaciones". En el siguiente ejercicio veremos cómo obtener una estructura como esta.

1. En `AlmacenPuntuacionesGson` añade la siguiente clase:

```

public class Clase {
    private ArrayList<Puntuacion> puntuaciones = new ArrayList<>();
    private boolean guardado;
}

```

2. Reemplaza el código subrayado de los siguientes métodos:

```

private Type type = new TypeToken<Clase>() {}.getType();

@Override
public void guardarPuntuacion(int puntos, String nombre, long fecha) {
    //string = leerString();
    Clase objeto;
    if (string == null) {
        objeto = new Clase();
    } else {
        objeto = gson.fromJson(string, type);
    }
}

```



```

    objeto.puntuaciones.add(new Puntuacion(puntos, nombre, fecha));
    string = gson.toJson(objeto, type);
    //guardarString(string);
}

@Override
public List<String> listaPuntuaciones(int cantidad) {
    //string = leerString();
    Clase objeto;
    if (string == null) {
        objeto = new Clase();
    } else {
        objeto = gson.fromJson(string, type);
    }
    List<String> salida = new ArrayList<>();
    for (Puntuacion puntuacion : objeto.puntuaciones) {
        salida.add(puntuacion.getPuntos()+" "+puntuacion.getNombre());
    }
    return salida;
}

```

3. Ejecuta el proyecto y verifica su funcionamiento. Si visualizas el valor de `string` este ha de ser:

```

{"guardado":false,
 "puntuaciones":[{"fecha":1478552190154,"nombre":"Mi nombre", "puntos":45000},
                  {"fecha":1478552205944,"nombre":"Otro nombre","puntos":31000}
 ]
}

```

Los saltos de línea han sido introducidos para facilitar la visualización.



Enlace de interés: Si te pasan un fichero JSON puede ser complejo crear el POJO adecuado para leerlo, especialmente si tiene objetos dentro de objetos, incluso con varios niveles de anidación. Este proceso puede realizarse de forma automática usando la herramienta <http://www.jsonschema2pojo.org/>

9.6.2. Procesando JSON con la librería org.json

La librería `org.json` permite tanto codificar datos en formato JSON dentro de un `String`, como el proceso inverso. Una de sus ventajas es que esta librería ya se encuentra integrada en la plataforma Android.

Para trabajar con esta librería hay que realizar el proceso de conversión manualmente, insertando cada elemento de uno en uno. Esto puede darnos más trabajo que otras librerías como GSON pero, al no ser un proceso automático, vamos a poder realizarlo de forma personalizada. Por ejemplo, podremos elegir el orden en que se generan los datos.



Ejercicio: Guardar puntuaciones en JSON con la librería `org.json`.

1. Crea la clase `AlmacenPuntuacionesJSON` con el siguiente código:

```

public class AlmacenPuntuacionesJSON implements AlmacenPuntuaciones {
    private String string; //Almacena puntuaciones en formato JSON

    public AlmacenPuntuacionesJSON() {
        guardarPuntuacion(45000,"Mi nombre", System.currentTimeMillis());
        guardarPuntuacion(31000,"Otro nombre", System.currentTimeMillis());
    }

    @Override
    public void guardarPuntuacion(int puntos, String nombre, long fecha) {

```

```

    //string = LeerString();
    List<Puntuacion> puntuaciones = leerJSon(string);
    puntuaciones.add(new Puntuacion(puntos, nombre, fecha));
    string = guardarJSon(puntuaciones);
    //guardarString(string);
}

@Override
public List<String> listaPuntuaciones(int cantidad) {
    //string = LeerFichero();
    List<Puntuacion> puntuaciones = leerJSon(string);
    List<String> salida = new ArrayList<>();
    for (Puntuacion puntuacion: puntuaciones) {
        salida.add(puntuacion.getPuntos()+" "+puntuacion.getNombre());
    }
    return salida;
}

private String guardarJSon(List<Puntuacion> puntuaciones) {
    String string = "";
    try {
        JSONArray jsonArray = new JSONArray();
        for (Puntuacion puntuacion : puntuaciones) {
            JSONObject objeto = new JSONObject();
            objeto.put("puntos", puntuacion.getPuntos());
            objeto.put("nombre", puntuacion.getNombre());
            objeto.put("fecha", puntuacion.getFecha());
            jsonArray.put(objeto);
        }
        string = jsonArray.toString();
    } catch (JSONException e) {
        e.printStackTrace();
    }
    return string;
}

private List<Puntuacion> leerJSon(String string) {
    List<Puntuacion> puntuaciones = new ArrayList<>();
    try {
        JSONArray json_array = new JSONArray(string);
        for (int i = 0; i < json_array.length(); i++) {
            JSONObject objeto = json_array.getJSONObject(i);
            puntuaciones.add(new Puntuacion(objeto.getInt("puntos"),
                objeto.getString("nombre"), objeto.getLong("fecha")));
        }
    } catch (JSONException e) {
        e.printStackTrace();
    }
    return puntuaciones;
}
}

```

2. Modifica el código necesario para que se pueda seleccionar este tipo de almacenamiento.
3. Si has realizado la práctica anterior, introduce los métodos `guardarString()` y `leerString()`.
4. Ejecuta el proyecto y verifica su funcionamiento.

NOTA: Acabamos de ver dos alternativas para serializar los datos contenidos en un objeto, XML y JSON. Existe otra alternativa aportada por el lenguaje Java, que consiste en implementar la interfaz `Serializable`¹. Aunque resulta muy sencillo de utilizar, también presenta algunos inconvenientes: La serialización ocupa mucho espacio (demasiado para transacciones por Internet), el formato obtenido es binario (no es visible o editable por un usuario) y solo se implementa en Java (no podemos interoperar con servidores con otros lenguajes).

¹ http://chuwiki.chuidiang.org/index.php?title=Serializaci%C3%B3n_de_objetos_en_java



Preguntas de repaso: *Trabajando con JSON*

9.7. Bases de datos con SQLite

Las bases de datos son una herramienta de gran potencia en la creación de aplicaciones informáticas. Hasta hace muy poco resultaba costoso y complejo utilizar bases de datos en nuestras aplicaciones. No obstante, Android incorpora la librería SQLite, que nos permitirá utilizar bases de datos mediante el lenguaje SQL, de una forma sencilla y utilizando muy pocos recursos del sistema. Como verás en este apartado, almacenar tu información en una base de datos no es mucho más complejo que almacenarla en un fichero, y además resulta mucho más potente.

SQL es el lenguaje de programación más utilizado para bases de datos. No resulta complejo entender los ejemplos que se mostrarán en este libro. No obstante, si deseas hacer cosas más complicadas te recomiendo que consultes alguno de los muchos manuales que se han escrito sobre el tema.

Para manipular una base de datos en Android usaremos la clase abstracta `SQLiteOpenHelper`, que nos facilita tanto la creación automática de la base de datos como el trabajar con futuras versiones de esta base de datos. Para crear un descendiente de esta clase hay que implementar los métodos `onCreate()` y `onUpgrade()`, y opcionalmente, `onOpen()`. La gran ventaja de utilizar esta clase es que ella se preocupará de abrir la base de datos si existe, o de crearla si no existe. Incluso de actualizar la versión si decidimos crear una nueva estructura de la base de datos. Además, esta clase tiene dos métodos: `getReadableDatabase()` y `getWritableDatabase()`, que abren la base de datos en modo solo lectura o lectura y escritura. En caso de que todavía no exista la base de datos, estos métodos se encargarán de crearla.



Vídeo[tutorial]: *Uso de bases de datos en Android*



Ejercicio: *Utilizando una base de datos para guardar puntuaciones*

Pasemos a demostrar cómo guardar las puntuaciones obtenidas en Asteroides en una base de datos. Si comparas la solución propuesta con las anteriores, verás que el código necesario es menor. Además, una base de datos te da mucha más potencia; puedes, por ejemplo, ordenar la salida por puntuación, eliminar filas antiguas, etc. Todo esto sin aumentar apenas el uso de recursos.

1. Crea la clase `AlmacenPuntuacionesSQLite` en el proyecto Asteroides y escribe el siguiente código:

```
public class AlmacenPuntuacionesSQLite extends SQLiteOpenHelper
    implements AlmacenPuntuaciones{
    public AlmacenPuntuacionesSQLite(Context context) {
        super(context, "puntuaciones", null, 1);
    }

    //Métodos de SQLiteOpenHelper
    @Override public void onCreate(SQLiteDatabase db) {
        db.execSQL("CREATE TABLE puntuaciones (" +
            "_id INTEGER PRIMARY KEY AUTOINCREMENT, " +
            "puntos INTEGER, nombre TEXT, fecha BIGINT)");
    }

    @Override public void onUpgrade(SQLiteDatabase db,
        int oldVersion, int newVersion) {
        // En caso de una nueva versión habría que actualizar las tablas
    }
}
```

```

}

//Métodos de AlmacenPuntuaciones
public void guardarPuntuacion(int puntos, String nombre,
                               long fecha) {
    SQLiteDatabase db = getWritableDatabase();
    db.execSQL("INSERT INTO puntuaciones VALUES ( null, "+
        puntos+", '"+nombre+"', "+fecha+"");
    db.close();
}

public List<String> listaPuntuaciones(int cantidad) {
    List<String> result = new ArrayList<String>();
    SQLiteDatabase db = getReadableDatabase();
    Cursor cursor = db.rawQuery("SELECT puntos, nombre FROM " +
        "puntuaciones ORDER BY puntos DESC LIMIT " + cantidad, null);
    while (cursor.moveToNext()){
        result.add(cursor.getInt(0)+" " +cursor.getString(1));
    }
    cursor.close();
    db.close();
    return result;
}
}

```

El constructor de la clase se limita a llamar al constructor heredado con el perfil:

```

SQLiteOpenHelper(Context contexto, String nombre, SQLiteDatabase.CursorFactory
    cursor, int version).

```

Los parámetros se describen a continuación:

contexto: Contexto usado para abrir o crear la base de datos.

nombre: Nombre de la base de datos que se creará. En nuestro caso, "puntuaciones".

cursor: Se utiliza para crear un objeto de tipo cursor. En nuestro caso no lo necesitamos.

version: Número de versión de la base de datos empezando desde 1. En el caso de que la base de datos actual tenga una versión más antigua se llamará a `onUpgrade()` para que actualice la base de datos.

El método `onCreate()` se invocará cuando sea necesario crear la base de datos. Como parámetro se nos pasa una instancia de la base de datos que se acaba de crear. Este es el momento de crear las tablas que contendrán información. En nuestra aplicación necesitamos solo la tabla `puntuaciones`, que se crea por medio del comando SQL `CREATE TABLE puntuaciones...` El primer campo tiene por nombre `_id` y será un entero usado como clave principal. Su valor será introducido de forma automática por el sistema, de forma que dos registros no tengan nunca el mismo valor.

El método `onUpgrade()` está vacío. Si más adelante decidimos crear una nueva estructura para la base de datos, tendremos que indicar un número de versión superior, por ejemplo la 2. Cuando se ejecute el código sobre un sistema donde se disponga de una base de datos con la versión 1, se invocará el método `onUpgrade()`. En él tendremos que escribir los comandos necesarios para transformar la antigua base de datos en la nueva, tratando de conservar la información de la versión anterior.

Pasemos a describir los dos métodos de la interfaz `AlmacenaPuntuaciones`. El método `guardarPuntuacion()` comienza obteniendo una referencia a nuestra base de datos utilizando `getWritableDatabase()`, mediante la cual ejecuta el comando SQL para almacenar un nueva fila en la tabla `INSERT INTO puntuaciones...` El método `listaPuntuaciones()` comienza obteniendo una referencia a nuestra base de datos utilizando `getReadableDatabase()`. Realiza una consulta utilizando el método `rawQuery()`, con la que obtiene un cursor que utiliza para leer todas las filas devueltas en la consulta.

2. Modifica el código correspondiente para que este método pueda ser seleccionado para almacenar las puntuaciones.
3. Verifica su funcionamiento.



Ejercicio: Verificación de los ficheros creados

1. Selecciona la pestaña *Device File Explorer* (esquina inferior derecha).
2. Busca la ruta *data/data/org.example.asteroides*.
3. Observa los ficheros creados y compara el tamaño de cada uno.

Name	Size	Date	Time	Permissions
org.example.asteroides		2011-07-27	22:24	drwxr-xr-
databases		2011-09-23	10:32	drwxrwx--
puntuaciones	5120	2011-10-15	07:51	-rw-rw--
files		2011-09-23	10:03	drwxrwx--
puntuaciones.txt	28	2011-09-23	10:05	-rw-rw--
lib		2011-07-27	22:24	drwxr-xr-
shared_prefs		2011-09-21	12:14	drwxrwx--
org.example.asteroides_preferences.xml	215	2011-09-22	20:49	-rw-rw--

9.7.1. Los métodos *query()* y *rawQuery()*

En el ejemplo anterior hemos utilizado el método `rawQuery()` para hacer una consulta. Este método tiene una versión alternativa con la misma función: el método `query()`. El método `query()` es el usado por defecto en la documentación oficial y además es el único disponible en otras clases (por ejemplo, para hacer una consulta en un `ContentProvider`). Sin embargo, tiene un inconveniente respecto al método `rawQuery()`: has de rellenar una gran cantidad de parámetros para controlar la búsqueda, lo que lo hace confuso de utilizar. Si estás acostumbrado a trabajar con SQL, es posible que este método te resulte incómodo. A continuación se describen los parámetros de ambos métodos:

```
Cursor SQLiteDatabase.query (
    String table,           //tabla a consultar (FROM)
    String[] columns,       //columnas a devolver (SELECT)
    String selection,       //consulta (WHERE)
    String[] selectionArgs, //reemplaza "?" de la consulta
    String groupBy,         //agrupado por (GROUPBY)
    String having,          //condición para agrupación
    String orderBy,         //ordenado por
    String limit)           //cantidad máx. de registros

Cursor SQLiteDatabase.rawQuery(
    String sql,             //comando SQL
    String[] selectionArgs) //reemplaza "?" de la consulta
```

Veamos un ejemplo de cómo se podrían utilizar estos métodos. Supongamos que hemos creado la tabla `tabla` y que tiene las columnas `texto`, `entero` y `numero`. Si quisiéramos seleccionar las columnas `texto` y `entero` de las filas con el valor de `numero` mayor que 2, ordenados según el valor de `entero` y que además el número de filas seleccionadas estuviera limitado a un máximo de `cantidad` (donde `cantidad` ha de ser una variable de tipo entero previamente definida), escribiríamos:

```
Cursor cursor = db.rawQuery("SELECT texto, entero FROM tabla" +
    " WHERE numero>2 ORDER BY entero LIMIT " + cantidad, null);
```

Cuando uno está acostumbrado al lenguaje SQL, esta puede ser la forma más sencilla de hacer la consulta. De forma alternativa podemos hacer uso del segundo parámetro. Este ha de ser un `array` de `String`, de forma que estos `strings` reemplacen cada una de las apariciones del carácter `"?"` en la cadena del primer parámetro. Veamos un ejemplo que sería equivalente al anterior:

```
String[] param = new String[1];
param[0] = Integer.toString(cantidad,10);
Cursor cursor = db.rawQuery("SELECT texto, entero FROM tabla" +
    " WHERE numero>2 ORDER BY entero LIMIT ?", param);
```

Si en lugar del método `rawQuery()` queremos utilizar el método `query()`, usaríamos el siguiente código equivalente a los dos anteriores:

```
String[] CAMPOS = {"texto", "entero"};
Cursor cursor = db.query("tabla", CAMPOS, "numero>2", null,
    null, null, "entero", Integer.toString(cantidad));
```



Ejercicio: Utilización del método `query()` para guardar puntuaciones

1. Reemplaza la llamada al método `rawQuery()` del ejercicio anterior por el siguiente código:

```
String[] CAMPOS = {"puntos", "nombre"};
Cursor cursor=db.query("puntuaciones", CAMPOS, null, null,
    null, null, "puntos DESC", Integer.toString(cantidad));
```

2. Verifica que el funcionamiento es idéntico.



Preguntas de repaso: SQLite I

9.7.2. Uso de bases de datos en Mis Lugares

En los próximos ejercicios pasamos a demostrar cómo guardar los datos de la aplicación Mis Lugares en una base de datos. Esta estará formada por una única tabla (`lugares`). A continuación, se muestran las columnas que contendrán y las filas que se introducirán como ejemplo. Los valores que aparecen en las columnas `_id` y `fecha` no coincidirán con los valores reales:

<code>_id</code>	<code>nombre</code>	<code>direccion</code>	<code>longitud</code>	<code>latitud</code>	<code>tipo</code>	<code>foto</code>	<code>telefono</code>	<code>url</code>	<code>Comentario</code>	<code>fecha</code>	<code>valoracion</code>
1	Escuela	C/ Paran	-0.166	38.99	7		962849	ht	Uno de lo	2345	3.0
2	Al de	P. Indust	-0.190	38.92	2		636472	ht	No te pier	2345	3.0
4	android	ciberesp	0.0	0.0	7			ht	Amplia tu	2345	5.0
7	Barranc	Vía Verd	-0.295	38.86	9			ht	Espectacu	2345	4.0
5	La Vital	Avda. de	-0.172	38.97	6		962881	ht	El típico c	2345	2.0

Tabla 1: Estructura de la tabla `lugares` de la base de datos `Lugares`.



Ejercicio: Utilizando una base de datos en Mis Lugares

1. Comenzamos haciendo una copia del proyecto, dado que en la nueva versión se eliminará parte del código desarrollado y es posible que quieras consultarlo en un futuro. Abre en el explorador de ficheros la carpeta que contiene el proyecto. Para hacer esto, puedes pulsar con el botón derecho sobre `app` en el explorador del proyecto y seleccionar `Show in Explorer`. Haz una copia de esta carpeta con un nuevo nombre y abre el nuevo proyecto.
2. Crea la clase `LugaresBD` en el proyecto y escribe el siguiente código:

```
public class LugaresBD extends SQLiteOpenHelper {

    Context contexto;

    public LugaresBD(Context contexto) {
        super(contexto, "lugares", null, 1);
        this.contexto = contexto;
    }

    @Override public void onCreate(SQLiteDatabase bd) {
        bd.execSQL("CREATE TABLE lugares (" +
            "_id INTEGER PRIMARY KEY AUTOINCREMENT, " +
            "nombre TEXT, " +
```

```

        "direccion TEXT, " +
        "longitud REAL, " +
        "latitud REAL, " +
        "tipo INTEGER, " +
        "foto TEXT, " +
        "telefono INTEGER, " +
        "url TEXT, " +
        "comentario TEXT, " +
        "fecha BIGINT, " +
        "valoracion REAL)");
    bd.execSQL("INSERT INTO lugares VALUES (null, "+
        "'Escuela Politécnica Superior de Gandía', "+
        "'C/ Paraninf, 1 46730 Gandia (SPAIN)', -0.166093, 38.995656, "+
        TipoLugar.EDUCACION.ordinal() + ", '', 962849300, "+
        "'http://www.epsg.upv.es', "+
        "'Uno de los mejores lugares para formarse.', "+
        System.currentTimeMillis() + ", 3.0)");
    bd.execSQL("INSERT INTO lugares VALUES (null, 'Al de siempre', "+
        "'P.Industrial Junto Molí Nou - 46722, Benifla (Valencia)', "+
        "-0.190642, 38.925857, " + TipoLugar.BAR.ordinal() + ", '', "+
        "636472405, '', '"+ "No te pierdas el arroz en calabaza.'", "+
        System.currentTimeMillis() + ", 3.0)");
    bd.execSQL("INSERT INTO lugares VALUES (null, 'androidcurso.com', "+
        "'ciberespacio', 0.0, 0.0, "+TipoLugar.EDUCACION.ordinal()+", '', "+
        "962849300, 'http://androidcurso.com', "+
        "'Amplia tus conocimientos sobre Android.', "+
        System.currentTimeMillis() + ", 5.0)");
    bd.execSQL("INSERT INTO lugares VALUES (null, 'Barranco del Infierno', "+
        "'Vía Verde del río Serpis. Villalonga (Valencia)', -0.295058, "+
        "38.867180, "+TipoLugar.NATURALEZA.ordinal() + ", '', 0, "+
        "'http://sosegaos.blogspot.com.es/2009/02/lorcha-villalonga-via-verde-del-"+
        "rio.html', 'Espectacular ruta para bici o andar', "+
        System.currentTimeMillis() + ", 4.0)");
    bd.execSQL("INSERT INTO lugares VALUES (null, 'La Vital', "+
        "'Avda. La Vital,0 46701 Gandia (Valencia)',-0.1720092,38.9705949, "+
        TipoLugar.COMPRAS.ordinal() + ", '', 962881070, "+
        "'http://www.lavital.es', 'El típico centro comercial', "+
        System.currentTimeMillis() + ", 2.0)");
}

@Override public void onUpgrade(SQLiteDatabase db, int oldVersion,
                                int newVersion) {
}
}

```

```

class LugaresBD(val contexto: Context) :
    SQLiteOpenHelper(contexto, "lugares", null, 1) {
    override fun onCreate(db: SQLiteDatabase) {
        bd.execSQL("CREATE TABLE lugares (" +
            "_id INTEGER PRIMARY KEY AUTOINCREMENT, " +
            "nombre TEXT, " +
            "direccion TEXT, " +
            "longitud REAL, " +
            "latitud REAL, " +
            "tipo INTEGER, " +
            "foto TEXT, " +
            "telefono INTEGER, " +
            "url TEXT, " +
            "comentario TEXT, " +
            "fecha BIGINT, " +
            "valoracion REAL)")
        bd.execSQL(("INSERT INTO lugares VALUES (null, " +
            "'Escuela Politécnica Superior de Gandía', " +
            "'C/ Paraninf, 1 46730 Gandia (SPAIN)', -0.166093, 38.995656, "+
            TipoLugar.EDUCACION.ordinal() + ", '', 962849300, "+
            "'http://www.epsg.upv.es', " +

```



```

        "Uno de los mejores lugares para formarse.', " +
        System.currentTimeMillis() + ", 3.0"))
    bd.execSQL(("INSERT INTO lugares VALUES (null, 'Al de siempre', " +
        "'P.Industrial Junto Molí Nou - 46722, Benifla (Valencia)', " +
        "-0.190642, 38.925857, " + TipoLugar.BAR.ordinal + ", '', " +
        "636472405, '', " + "'No te pierdas el arroz en calabaza.', " +
        System.currentTimeMillis() + ", 3.0"))
    bd.execSQL(("INSERT INTO lugares VALUES (null, 'androidcurso.com', " +
        "'ciberespacio', 0.0, 0.0,"+TipoLugar.EDUCACION.ordinal+", '', " +
        "962849300, 'http://androidcurso.com', " +
        "'Amplia tus conocimientos sobre Android.', " +
        System.currentTimeMillis() + ", 5.0"))
    bd.execSQL(("INSERT INTO lugares VALUES (null,'Barranco del Infierno','+
        "'Vía Verde del río Serpis. Villalonga (Valencia)', -0.295058, "+
        "38.867180, " + TipoLugar.NATURALEZA.ordinal + ", '', 0, " +
        "'http://sosegaos.blogspot.com.es/2009/02/lorcha-villalonga-via-verde-del-"+
        "rio.html', 'Espectacular ruta para bici o andar', " +
        System.currentTimeMillis() + ", 4.0"))
    bd.execSQL(("INSERT INTO lugares VALUES (null, 'La Vital', " +
        "'Avda. La Vital,0 46701 Gandia (Valencia)',-0.1720092,38.9705949,"+
        TipoLugar.COMPRAS.ordinal + ", '', 962881070, " +
        "'http://www.lavital.es', 'El típico centro comercial', " +
        System.currentTimeMillis() + ", 2.0"))
}

override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int,
                        newVersion: Int) {}
}

```

El constructor de la clase se limita a llamar al constructor heredado. Los parámetros se describen a continuación:

contexto: Contexto usado para abrir o crear la base de datos.

nombre: Nombre de la base de datos que se creará. En nuestro caso, “puntuaciones”.

cursor: Se utiliza para crear un objeto de tipo cursor. En nuestro caso no lo necesitamos.

version: Número de versión de la base de datos empezando desde 1. En el caso de que la base de datos actual tenga una versión más antigua se llamará a `onUpgrade()` para que actualice la base de datos.

El método `onCreate()` se invocará cuando sea necesario crear la base de datos. Como parámetro se nos pasa una instancia de la base de datos que se acaba de crear. Este es el momento de crear las tablas que contendrán información. El primer campo tiene por nombre `_id` y será un entero usado como clave principal. Su valor será introducido de forma automática por el sistema, de forma que dos registros no tengan nunca el mismo valor.

En nuestra aplicación necesitamos solo la tabla `lugares`, que es creada por medio del comando SQL `CREATE TABLE lugares...`. La primera columna tiene por nombre `_id` y será un entero usado como clave principal. Su valor será introducido automáticamente por el sistema, de forma que dos filas no tengan nunca el mismo valor de `_id`.

Las siguientes líneas introducen nuevas filas en la tabla utilizando el comando SQL `INSERT INTO lugares VALUES (, , ...)`. Los valores deben introducirse en el mismo orden que las columnas. La primera columna se deja como `null` dado que corresponde al `_id` y es el sistema quien ha de averiguar el valor correspondiente. Los valores de tipo `TEXT` deben introducirse entre comillas, pudiendo utilizar comillas dobles o simples. Como en Java se utilizan comillas dobles, en SQL utilizaremos comillas sencillas. El valor `TipoLugar.EDUCACION.ordinal()` corresponde a un entero según el orden en la definición de este enumerado y `System.currentTimeMillis()` corresponde a la fecha actual representada como número de milisegundos transcurridos desde 1970. El resto de los valores son sencillos de interpretar.

Ha de quedar claro que este constructor solo creará una base de datos (llamando a `onCreate()`) si todavía no existe. Si ya fue creada en una ejecución anterior, nos devolverá la base de datos existente.

El método `onUpgrade()` está vacío. Si más adelante, en una segunda versión de Mis Lugares, decidiéramos crear una nueva estructura para la base de datos, tendríamos que indicar un número de versión superior, por ejemplo, la 2. Cuando se ejecute el código sobre un sistema que disponga de una base de datos con la versión 1, se invocará el método `onUpgrade()`. En él tendremos que escribir los comandos necesarios para transformar la antigua base de datos en la nueva, tratando de conservar la información de la versión anterior.

- Para acceder a los datos de la aplicación se definió la interfaz `Lugares`. Vamos a implementar esta interfaz para que los cambios sean los mínimos posibles. Añade el texto subrayado a la clase:

```
public class LugaresBD extends SQLiteOpenHelper
    implements RepositorioLugares {
```

```
class LugaresBD(val contexto: Context) :
    SQLiteOpenHelper(contexto, "lugares", null, 1), RepositorioLugares {
```

Aparecerá un error justo en la línea que acabas de introducir. Si sitúas el cursor de texto sobre el error, aparecerá una bombilla roja con opciones para resolver el error. Pulsa en *“Implement methods”*, selecciona todos los métodos y pulsa OK. Observa cómo en la clase se añaden todos los métodos de esta interfaz. De momento vamos a dejar estos métodos sin implementar. En la sección “Operaciones con bases de datos en Mis Lugares” aprenderemos a realizar las operaciones básicas cuando trabajemos con datos: altas, bajas, modificaciones y consultas.

- No ejecutes todavía la aplicación. Hasta que no hagamos el siguiente ejercicio no funcionará correctamente.

9.7.3. Adaptadores para bases de datos

Un adaptador (*Adapter*) es un mecanismo de Android que hace de puente entre nuestros datos y las vistas contenidas en un `RecyclerView`, `ListView`, `GridView` o `Spinner`.



En el siguiente ejercicio vamos a crear un adaptador que toma la información de la base de datos que acabamos de crear y se la muestra a un `RecyclerView`. Realmente podríamos usar el adaptador `AdaptadorLugares` que ya tenemos creado. Este adaptador toma la información de un objeto que sigue la interfaz `Lugares`, restricción que cumple la clase `LugaresBD`. No obstante, vamos a realizar una implementación alternativa. La razón es que la implementación actual de `AdaptadorLugares` necesitaría una consulta a la base de datos cada vez que requiera una información de `Lugares`. (Veremos más adelante que cada llamada a `elemento()`, `añade()`, `nuevo()`... va a suponer un acceso a la base de datos.)

El nuevo adaptador, `AdaptadorLugaresBD`, va a trabajar de una forma más eficiente. Vamos a realizar una consulta de los elementos a listar y los va a guardar en un objeto de la clase `Cursor`. Mantendrá esta información mientras no cambie la información a listar, por lo que solo va a necesitar una consulta a la base de datos.



Ejercicio: Un Adaptador para base de datos en Mis Lugares

1. Crea la clase `AdaptadorLugaresBD` con el siguiente código:

```
public class AdaptadorLugaresBD extends AdaptadorLugares {

    protected Cursor cursor;

    public AdaptadorLugaresBD(RepositorioLugares lugares, Cursor cursor) {
        super(lugares);
        this.cursor = cursor;
    }

    public Cursor getCursor() {
        return cursor;
    }

    public void setCursor(Cursor cursor) {
        this.cursor = cursor;
    }

    public Lugar lugarPosicion(int posicion) {
        cursor.moveToPosition(posicion);
        return LugaresBD.extraeLugar(cursor);
    }

    public int idPosicion(int posicion) {
        cursor.moveToPosition(posicion);
        if (cursor.getCount() > 0) return cursor.getInt(0);
        else return -1;

        return cursor.getInt(0);
    }

    @Override
    public void onBindViewHolder(ViewHolder holder, int posicion) {
        Lugar lugar = lugarPosicion(posicion);
        holder.personaliza(lugar);
        holder.itemView.setTag(new Integer(posicion));
    }

    @Override public int getItemCount() {
        return cursor.getCount();
    }
}
```

```
class AdaptadorLugaresBD(lugares: LugaresBD, var cursor: Cursor)
    : AdaptadorLugares(lugares){

    fun lugarPosicion(posicion: Int): Lugar {
        cursor.moveToPosition(posicion)
        return (lugares as LugaresBD).extraeLugar(cursor)
    }

    fun idPosicion(posicion: Int): Int {
        cursor.moveToPosition(posicion)
        if (cursor.count > 0) return cursor.getInt(0)
        else return -1
    }

    override fun onBindViewHolder(holder: AdaptadorLugares.ViewHolder,
                                   posicion: Int) {

        val lugar = lugarPosicion(posicion)
        holder.personaliza(lugar, onClick)
        holder.view.tag = posicion
    }
}
```

```

override fun getItemCount(): Int {
    return cursor.getCount()
}
}

```

Esta clase extiende `AdaptadorLugares`; de esta forma aprovechamos la mayor parte del código del adaptador y solo tenemos que indicar las diferencias. La más importante es que ahora el constructor tiene un nuevo parámetro de tipo `Cursor`, que es el resultado de una consulta en la base de datos. Realmente es aquí donde vamos a buscar los elementos a listar en el `RecyclerView`. Esta forma de trabajar es mucho más versátil que utilizar un `array`, podemos listar un tipo de lugar o que cumplan una determinada condición sin más que realizar un pequeño cambio en la consulta SQL. Además, podremos ordenarlos por valoración o cualquier otro criterio, porque se mostrarán en el mismo orden como aparecen en el `Cursor`. Por otra parte, resulta muy eficiente dado que se realiza solo una consulta a la base de datos, dejando el resultado almacenado en la variable de tipo `Cursor`.

En Java el constructor de la clase se limita a llamar al `super()` y a almacenar el nuevo parámetro en una variable global. En Kotlin este proceso se indica en la declaración. En Java se han añadido los métodos `getter` y `setter` que permiten acceder al `Cursor` desde fuera de la clase.

Con el método `lugarPosicion()` vamos a poder acceder a un lugar, indicar su posición en `Cursor` o, lo que es lo mismo, su posición en el listado. Para ello, movemos el cursor a la posición indicada y extraemos el lugar en esta posición, utilizando un método estático de `LugarBD`.

Cuando queramos realizar una operación de borrado o edición de un registro de la base de datos, vamos a identificar el lugar a modificar por medio de la columna `_id`. Recuerda que esta columna ha sido definida en la posición 0. Para obtener el `id` de un lugar conociendo la posición que ocupa en el listado, se ha definido el método `lugarPosicion()`.

Los dos últimos métodos ya existen en la clase que extendemos, pero los vamos a reemplazar; por esta razón tienen la etiqueta de `override`. El primero es `onBindViewHolder()` que se utilizaba para personalizar la vista `ViewHolder` en una determinada posición. La gran diferencia entre el nuevo método es que ahora el lugar lo obtenemos del cursor, mientras que en el método anterior se obtenía de `lugares`. Esto supondría una nueva consulta en la base de datos por cada llamada a `onBindViewHolder()`, lo que sería muy poco eficiente. En el método `getItemCount()` pasa algo similar, obtener el número de elementos directamente del cursor es más eficiente que hacer una nueva consulta.

Observa la última línea de `onBindViewHolder()` (`holder.view.tag = posicion`). El atributo `Tag` permite asociar a una vista cualquier objeto con información extra. La idea es asociar a cada vista del `RecyclerView` la posición que ocupa en el listado. La idea es que cuando asociamos un `onClickListener` este nos indica la vista pulsada, pero no la posición. De esta forma, sabiendo la vista conoceremos su posición. En la implementación anterior usábamos un método alternativo: `posición = recyclerView.getChildAdapterPosition(vista)`. Pero tiene el inconveniente de necesitar el `recyclerView`. Y ahora no vamos a disponer de él.

En Kotlin tanto las clases como los atributos son por defecto cerrados². Por lo tanto, aparece un error al intentar heredar de `AdaptadorLugares`. Para resolverlo pulsa sobre la bombilla roja y selecciona `Make AdaptadorLugares Open`.

2. Añade a la clase `LugaresBD` los siguientes métodos:

```

public static Lugar extraeLugar(Cursor cursor) {
    Lugar lugar = new Lugar();
    lugar.setNombre(cursor.getString(1));
    lugar.setDireccion(cursor.getString(2));
    lugar.setPosicion(new GeoPunto(cursor.getDouble(3),
        cursor.getDouble(4)));
    lugar.setTipo(TipoLugar.values()[cursor.getInt(5)]);
    lugar.setFoto(cursor.getString(6));
    lugar.setTelefono(cursor.getInt(7));
    lugar.setUrl(cursor.getString(8));
    lugar.setComentario(cursor.getString(9));
}

```

² <https://youtu.be/sMbrh3-1ufA>

```

    lugar.setFecha(cursor.getLong(10));
    lugar.setValoracion(cursor.getFloat(11));
    return lugar;
}

public Cursor extraeCursor() {
    String consulta = "SELECT * FROM lugares";
    SQLiteDatabase bd = getReadableDatabase();
    return bd.rawQuery(consulta, null);
}

```

```

fun extraeLugar(cursor: Cursor) = Lugar(
    nombre = cursor.getString(1),
    direccion = cursor.getString(2),
    posicion = GeoPunto(cursor.getDouble(3), cursor.getDouble(4)),
    tipoLugar = TipoLugar.values()[cursor.getInt(5)],
    foto = cursor.getString(6),
    telefono = cursor.getInt(7),
    url = cursor.getString(8),
    comentarios = cursor.getString(9),
    fecha = cursor.getLong(10),
    valoracion = cursor.getFloat(11) )

fun extraeCursor(): Cursor =
    readableDatabase.rawQuery("SELECT * FROM lugares", null)

```

El primer método crea un nuevo lugar con los datos de la posición actual de un **Cursor**. El segundo nos devuelve un cursor que contiene todos los datos de la tabla.

3. Abre la clase **Aplicacion** y reemplaza la declaración de la variable **lugares** y añade adaptador:

```

public LugaresBD lugares;
public AdaptadorLugaresBD adaptador;

@Override public void onCreate() {
    super.onCreate();
    lugares = new LugaresBD(this);
    adaptador= new AdaptadorLugaresBD(lugares, lugares.extraeCursor());
}

```

```

val lugares = LugaresBD(this)
val adaptador by lazy {AdaptadorLugaresBD(lugares, lugares.extraeCursor())}

```

Hemos pasado **adaptador** dentro de **Aplicacion** para que sea accesible desde cualquier clase. Ahora la lista de lugares que el usuario a buscado en la base de datos estará almacenada dentro de **adaptador**. Y queremos acceder a esta lista desde varios sitios. En **Kotlin** la inicialización de las propiedades se recomienda realizarla en su declaración. Observa como para **adaptador** se utiliza **by lazy**, para indicar que la inicialización se realice cuando vallamos a utilizar la variable. De hacerlo inmediatamente corremos el peligro de que la base de datos no esté creada.

4. Añade en **MainActivity** la siguiente propiedad:

```

private RepositorioLugaresBD lugares;
private AdaptadorLugaresBD adaptador;

@Override protected void onCreate(Bundle savedInstanceState) {
    ...
    adaptador = ((Aplicacion) getApplication()).adaptador;
}

```

```

val adaptador by lazy { (application as Aplicacion).adaptador }

```

5. Reemplaza en **MainActivity** dentro de **onCreate()** en código subrayado:

```
adaptador.setOnItemClickListener(new View.OnClickListener() {
    @Override public void onClick(View v) {
        int pos = (Integer)(v.getTag());
        usoLugar.mostrar(pos);
    }
});
```

```
adaptador.onClick = {
    val pos = it.tag as Int
    usoLugar.mostrar(pos)
}
```

6. Ejecuta la aplicación y verifica que la lista se muestra correctamente.
7. Si pulsas sobre un elemento del `RecyclerView` posiblemente se producirá un error. Para solucionarlo abre la clase `VistaLugarActivity` añade la propiedad `adaptador` y modifica `lugares` como hemos hecho en el punto 6 y reemplaza:

```
lugar = lugares.elemento adaptador.lugarPosicion (pos);
```

```
lugar = lugares.elemento adaptador.lugarPosicion (pos)
```

Recuerda que el método `elemento()` todavía no ha sido implementado. Además, como ya hemos comentado, resulta más eficiente acceder a este lugar usando la lista almacenada en `adaptador`.

8. En la clase `EdicionLugarActivity` realiza las mismas operaciones.
9. En la clase `CasosUsoLugar` pasaremos el `adaptador` en el constructor:

```
private RepositorioLugaresBD lugares;
private AdaptadorLugaresBD adaptador;

public CasosUsoLugar(Activity actividad,
    RepositorioLugaresBD lugaresBD lugares,
    AdaptadorLugaresBD adaptador) {
    ...
    this.adaptador = adaptador;
}
```

```
class CasosUsoLugar(
    val actividad: Activity,
    val lugares: RepositorioLugaresBD,
    val adaptador: AdaptadorLugaresBD)
```

10. Añade el nuevo parámetro en las llamadas a este constructor.
11. Ejecuta la aplicación y verifica que funciona. Puedes seleccionar un lugar e incluso editarlo, aunque si guardas una edición no se almacenarán los cambios. Lo arreglaremos más adelante.



Ejercicio: Adaptando la actividad del Mapa al nuevo adaptador

En este ejercicio adaptaremos la actividad `MapaActivity` para que use adecuadamente el nuevo adaptador basado en `Cursor`. El proceso es el mismo que acabamos de realizar: Reemplazaremos los accesos a `Aplicacion.lugares` por `Aplicacion.adaptador`.

1. Reemplaza inicialización de la variable `lugares` por `adaptador`:

```
private RepositorioLugaresBD lugares;
private AdaptadorLugaresBD adaptador;

@Override public void onCreate(Bundle savedInstanceState) {
    ...
    lugares = ((Aplicacion) getApplication()).lugares;
    adaptador = ((Aplicacion) getApplication()).adaptador;
```



```
val lugares by lazy { (application as Aplicacion).lugares }  
val adaptador by lazy { (application as Aplicacion).adaptador }
```

2. Reemplaza el código del método `onMapReady()`:

```
if (Lugares.tamaño() < adaptador.getItemCount() > 0) {  
    GeoPunto p= Lugares.elemento adaptador.lugarPosicion(0).getPosicion();  
    ...  
    for (int n=0; n< Lugares.tamaño() adaptador.getItemCount(); n++) {  
        Lugar lugar = Lugares.elemento adaptador.lugarPosicion(n);  
    }
```

```
if (lugares.tamaño() < adaptador.itemCount > 0) {  
    val p = lugares.elemento adaptador.lugarPosicion(0).posicion  
    ...  
    for (n in 0 until lugares.tamaño() adaptador.itemCount) {  
        val lugar = lugares.elemento adaptador.lugarPosicion(n)  
    }
```

3. En el método `onInfoWindowClick()` reemplaza:

```
for (int pos=0; pos< Lugares.tamaño() adaptador.getItemCount(); pos++){  
    if (Lugares.elemento adaptador.lugarPosicion(pos).getNombre()
```

```
for (pos in 0 until lugares.tamaño() adaptador.itemCount) {  
    val lugar = lugares.elemento adaptador.lugarPosicion(pos)
```

Como hemos indicado, la ventaja de este cambio es que no realizaremos nuevos accesos a la base de datos una vez obtenido el `Cursor`.

4. Verifica el funcionamiento de la actividad `MapaActivity`.



Práctica: Probando consultas en Mis Lugares

1. En el método `extraeCursor()` de la clase `LugaresBD` reemplaza el comando `SELECT * FROM lugares` por `SELECT * FROM lugares WHERE valoracion>1.0 ORDER BY nombre LIMIT 4`. Ejecuta la aplicación y verifica la nueva lista.
2. Realiza otras consultas similares. Si tienes dudas, puedes consultar en Internet la sintaxis del comando SQL `SELECT`.
3. Si quieres practicar el uso del método `query()`, puedes tratar de realizar una consulta utilizando este método.



Práctica: Añadir criterios de ordenación y máximo en Preferencias

1. Modifica el método `extraeCursor()` para que el criterio de ordenación y el máximo de lugares a mostrar corresponda con los valores que el usuario ha indicado en las preferencias.
2. Si el usuario escoge el primer criterio de ordenación has de dejar la consulta original sin introducir la cláusula `"ORDER BY"`.
3. Si escoge el orden por valoración este ha de ser descendiente, de más valorados a menos. Puedes usar la cláusula `"ORDER BY valoracion DESC"`.
4. Para ordenar por distancia puedes usar la siguiente consulta SQL:

```
"SELECT * FROM lugares ORDER BY " +  
    "(" + lon + "-longitud)*(" + lon + "-longitud) + " +  
    "(" + lat + "-latitud)*(" + lat + "-latitud )"
```

Donde las variables `lon` y `lat` han de corresponder con la posición actual del dispositivo. Esta ecuación es una simplificación que no tiene en cuenta que los polos están achatados, pero funciona de forma adecuada.

5. Si no actualizamos el cursor con la lista el cambio de preferencias no será efectivo hasta que salgas de aplicación y vuelvas a entrar. Para evitar este inconveniente, llama a la actividad `PreferenciasActivity` mediante `startActivityForResult()`. En el método `onActivityResult()` has de actualizar el cursor de `adaptado` e indicar todos los elementos han de redibujarse. Para esta última acción puedes utilizar `adaptador.notifyDataSetChanged()`.



Solución:

1. Reemplaza en `lugaresBD` el siguiente método:

```
public Cursor extraeCursor() {
    SharedPreferences pref =
        PreferenceManager.getDefaultSharedPreferences(contexto);
    String consulta;
    switch (pref.getString("orden", "0")) {
        case "0":
            consulta = "SELECT * FROM lugares ";
            break;
        case "1":
            consulta = "SELECT * FROM lugares ORDER BY valoracion DESC";
            break;
        default:
            double lon = ((Aplicacion) contexto.getApplicationContext())
                .posicionActual.getLongitude();
            double lat = ((Aplicacion) contexto.getApplicationContext())
                .posicionActual.getLatitude();
            consulta = "SELECT * FROM lugares ORDER BY " +
                "(" + lon + "-longitud)*(" + lon + "-longitud) + " +
                "(" + lat + "-latitud)*(" + lat + "-latitud)";
            break;
    }
    consulta += " LIMIT "+pref.getString("maximo","12");
    SQLiteDatabase bd = getReadableDatabase();
    return bd.rawQuery(consulta, null);
}
```

```
fun extraeCursor(): Cursor {
    val pref = PreferenceManager.getDefaultSharedPreferences(contexto)
    var consulta = when (pref.getString("orden", "0")) {
        "0" -> "SELECT * FROM lugares "
        "1" -> "SELECT * FROM lugares ORDER BY valoracion DESC"
        else -> {
            val lon = (contexto.getApplicationContext() as Aplicacion)
                .posicionActual.longitud
            val lat = (contexto.getApplicationContext() as Aplicacion)
                .posicionActual.latitud
            "SELECT * FROM lugares ORDER BY " +
                "($lon - longitud)*($lon - longitud) + " +
                "($lat - latitud)*($lat - latitud)"
        }
    }
    consulta += " LIMIT ${pref.getString("maximo", "12")}"
    return readableDatabase.rawQuery(consulta, null)
}
```

2. En `MainActivity` añade:

```
static final int RESULTADO_PREFERENCIAS = 0;

public void lanzarPreferencias(View view) {
    Intent i = new Intent(this, PreferenciasActivity.class);
    startActivityForResult(i, RESULTADO_PREFERENCIAS);
}
```

```

@Override protected void onActivityResult(int requestCode, int resultCode,
                                           Intent data) {
    if (requestCode == RESULTADO_PREFERENCIAS) {
        adaptador.setCursor(Lugares.extraeCursor());
        adaptador.notifyDataSetChanged();
    }
}

```

```

val RESULTADO_PREFERENCIAS = 0

fun lanzarPreferencias(view: View? = null) = startActivityForResult(
    Intent(this, PreferenciasActivity::class.java), RESULTADO_PREFERENCIAS)

override
fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    if (requestCode == RESULTADO_PREFERENCIAS) {
        adaptador.cursor = lugares.extraeCursor()
        adaptador.notifyDataSetChanged()
    }
}

```

Operaciones con bases de datos en Mis Lugares

En los apartados anteriores hemos aprendido a crear una base de datos y a realizar consultas en una tabla. En este apartado vamos a continuar aprendiendo las operaciones básicas cuando trabajamos con datos. Estas son: altas, bajas y modificaciones y consultas.



Ejercicio: Consulta de un elemento en Mis Lugares

- Reemplaza en la clase `LugaresBD` en el método `elemento()` con el siguiente código. Su finalidad es buscar el lugar correspondiente a un `id` y devolverlo.

```

@Override public Lugar elemento(int id) {
    Cursor cursor = getReadableDatabase().rawQuery(
        "SELECT * FROM lugares WHERE _id = "+id, null);
    try {
        if (cursor.moveToNext())
            return extraeLugar(cursor);
        else
            throw new SQLException("Error al acceder al elemento _id = "+id);
    } catch (Exception e) {
        throw e;
    } finally {
        if (cursor!=null) cursor.close();
    }
}

```

```

override fun elemento(id: Int): Lugar {
    val cursor = readableDatabase.rawQuery(
        "SELECT * FROM lugares WHERE _id = $id", null)
    try {
        if (cursor.moveToNext())
            return extraeLugar(cursor)
        else
            throw SQLException("Error al acceder al elemento _id = $id")
    } catch (e:Exception) {
        throw e
    } finally {
        cursor?.close()
    }
}

```

Comenzamos obteniendo la referencia a la base de datos con la propiedad `readableDatabase`. Este objeto nos permitirá hacer consultas en la base de datos. Por medio del método `rawQuery()` se realiza una consulta en la tabla `lugares` usando el comando SQL `SELECT * FROM lugares WHERE _id =...` Este comando podría interpretarse como “selecciona todas las columnas de la tabla `lugares`, para la fila con el `id` indicado”. El resultado de una consulta es un `Cursor` con la fila, si es encontrado, o en caso contrario, un `Cursor` vacío.

En la siguiente línea llamamos a `cursor.moveToNext()` para que el cursor pase a la siguiente fila encontrada. Como es la primera llamada estamos hablando del primer elemento. Devuelve `true` si lo encuentra y `false` si no. En caso de encontrarlo, llamamos a `extraerLugar()` para actualizar todos los atributos de `lugar` con los valores de la fila apuntada por el cursor. Si no lo encontramos lanzamos una excepción.

Es importante cerrar lo antes posibles el cursor por tener mucho consumo de memoria. Lo hacemos en la sección `finally` para asegurarnos que se realiza siempre, haya habido una excepción o no.

NOTA: Este método no es usado por la aplicación. Ha sido añadido por pertenecer a la interfaz `Lugares`.



Ejercicio: Modificación de un lugar

Si tratas de modificar cualquiera de los lugares, observarás que los cambios no tienen efecto. Para que la base de datos sea actualizada, realiza el siguiente ejercicio:

1. Añade en la clase `LugaresBD`, en el siguiente método `actualiza()`. Su finalidad es reemplazar el lugar correspondiente al `id` indicado por un nuevo lugar.

```
@Override public void actualiza(int id, Lugar lugar) {
    getWritableDatabase().execSQL("UPDATE lugares SET" +
        "    nombre = '" + lugar.getNombre() +
        "', direccion = '" + lugar.getDireccion() +
        "', longitud = '" + lugar.getPosicion().getLongitud() +
        "', latitud = '" + lugar.getPosicion().getLatitud() +
        "', tipo = '" + lugar.getTipo().ordinal() +
        "', foto = '" + lugar.getFoto() +
        "', telefono = '" + lugar.getTelefono() +
        "', url = '" + lugar.getUrl() +
        "', comentario = '" + lugar.getComentario() +
        "', fecha = '" + lugar.getFecha() +
        "', valoracion = '" + lugar.getValoracion() +
        " WHERE _id = " + id);
}
```

```
override fun actualiza(id:Int, lugar:Lugar) = with(lugar) {
    writableDatabase.execSQL("UPDATE lugares SET " +
        "nombre = '$nombre', direccion = '$direccion', " +
        "longitud = ${posicion.longitud}, latitud = ${posicion.latitud}, " +
        "tipo = ${tipoLugar.ordinal}, foto = '$foto', telefono = $telefono, " +
        "url = '$url', comentario = '$comentarios', fecha = $fecha, " +
        "valoracion = $valoracion WHERE _id = $id")
}
```

2. En la clase `EdicionLugarActivity`, en el método `onOptionsItemSelected()` añade el código subrayado y elimina el tachado:

```
int id = adaptador.idPosicion(pos);
usoLugar.guardar(pos _id, lugar);
```

```
val id = adaptador.idPosicion(pos)
usoLugar.guardar(pos _id, nuevoLugar)
```

La variable `pos` corresponde a un indicador de posición dentro de la lista. Para utilizar correctamente el método `actualiza()` de `LugaresBD`, hemos de obtener el `_id`

correspondiente a la primera columna de la tabla. Este cambio lo realiza el método `idPosicion()` de `adaptador`.

3. Ejecuta la aplicación y trata de modificar algún lugar. Observa que, cuando realizas un cambio en un lugar, estos no parecen en `VistaLugarActivity` ni en `MainActivity`. Realmente sí que se han almacenado, el problema está en que el `adaptador` no se ha actualizado. Para verificar que los cambios sí que se han almacenado, has de salir de la aplicación y volver a lanzarla.
4. Para resolver el refresco de `MainActivity` has de añadir la siguiente línea al final del método `guardar()` de `CasosUsoLugar`:

```
adaptador.setCursor(lugares.extraeCursor());
adaptador.notifyDataSetChanged();
```

```
adaptador.cursor = lugares.extraeCursor()
adaptador.notifyDataSetChanged()
```

Asignamos un nuevo cursor al adaptador y le indicamos que los datos han cambiado para que vuelva a crear las vistas correspondientes del `RecyclerView`.

5. Ejecuta de nuevo la aplicación. Tras editar un lugar, los cambios se reflejan en `MainActivity` pero no en `VistaLugarActivity`.
6. Para resolver este problema has de añadir las siguientes líneas en `VistaLugarActivity`:

```
public int id = -1;

@Override public void onActivityCreated(Bundle state) {
    ...
    if (extras != null) pos = extras.getInt("pos", 0);
    else pos = 0;
    id = adaptador.idPosicion(pos);
    ...
    @Override public void onActivityResult(int requestCode, int resultCode,...
    if (requestCode == RESULTADO_EDITAR) {
        lugar = lugares.elemento(id);
        pos = adaptador.posicionId(id);
        actualizaVistas();
    }
}
```

```
private var id: Int = -1

override fun onCreate(savedInstanceState: Bundle?) {
    ...
    pos = intent.extras?.getInt("pos", 0) ?: 0
    id = adaptador.idPosicion(pos)
    ...
    override fun onActivityResult(requestCode: Int, resultCode: Int, data: In...
    if (requestCode == RESULTADO_EDITAR) {
        lugar = lugares.elemento(id)
        pos = adaptador.posicionId(id)
        actualizaVistas()
    }
    ...
}
```

Necesitamos actualizar la variable `lugar` dado que esta acaba de ser modificada. Extraerla según su posición en el listado es potencialmente peligroso, dado que esta posición puede cambiar dinámicamente. Por ejemplo, si ordenamos los lugares por orden alfabético y modificamos su inicial, posiblemente cambie su posición. Por el contrario, el `_id` de un lugar nunca puede cambiar. Hemos obtenido el `_id` al crear la actividad. Tras la edición del lugar, con este `_id`, obtenemos los nuevos valores para `lugar` y buscamos la nueva posición a partir de `_id`.

7. Para hacer la última acción añade en `AdaptadorLugaresBD` la siguiente función:

```
public int posicionId(int id) {
    int pos = 0;
    while (pos < getItemCount() && idPosicion(pos) != id) pos++;
}
```

```

    if (pos >= getItemCount()) return -1;
    else return pos;
}

```

```

fun posicionId(id: Int): Int {
    var pos = 0
    while (pos < itemCount && idPosicion(pos) != id) pos++
    return if (pos >= itemCount) -1
           else pos
}

```

Como ves, se recorren todos los elementos del adaptador hasta encontrar uno con el mismo `id`. Si no es encontrado devolvemos -1.

8. Ejecuta la aplicación y verifica el nuevo funcionamiento.



Ejercicio: Modificación valoración y fotografía de un lugar

1. Algunos de los campos de un lugar no se modifican en la actividad `EdicionLugarActivity`, si no que se hacen directamente en `VistaLugarActivity`. En concreto la valoración, la fotografía y más adelante añadiremos fecha y hora. Cuando se modifiquen estos campos, también habrá que almacenarlos de forma permanente en la base de datos. Empezaremos por la valoración. Añade en el método `actualizaVistas()` de `VistaLugarActivity` el código subrayado.

```

valoracion.setOnRatingChangeListener(null);
valoracion.setRating(lugar.getValoracion());

@Override public void onRatingChanged(RatingBar ratingBar,
                                     float valor, boolean fromUser) {
    lugar.setValoracion(valor);
    usoLugar.actualizaPosLugar(pos, lugar);
    pos = adaptador.posicionId(id);
}

```

```

valoracion.setOnRatingChangeListener { _, _, _ -> }
valoracion.setRating(lugar.valoracion)
valoracion.setOnRatingChangeListener { _, valor, _ ->
    lugar.valoracion = valor
    usoLugar.actualizaPosLugar(pos, lugar)
    pos = adaptador.posicionId(id)
}

```

Cuando el usuario cambie la valoración de un lugar se llamará a `onRatingChanged()`, donde actualizamos la valoración y llamamos a `actualizarLugares()`. Esta función llamará a `actualizaVistas()`, donde cambiamos `raingBar`, lo que provocará una llamada al escuchador, y así sucesivamente entrando en bucle. Para evitarlo antes de cambiar el valor desactivamos el escuchador. Si tenemos seleccionada la ordenación por valoración, al cambiarla puede cambiar la posición del lugar en la lista. Por si ha cambiado, volvemos a obtener la variable `pos`.

2. Añade la siguiente función a `CasosUsoLugar`:

```

public void actualizaPosLugar(int pos, Lugar lugar) {
    int id = adaptador.idPosicion(pos);
    guardar(id, lugar);
}

```

```

fun actualizaPosLugar(pos: Int, lugar: Lugar) {
    val id = adaptador.idPosicion(pos)
    guardar(id, lugar);
}

```

Primero obtenemos en la variable `id` el identificador del lugar. Para ello, vamos a usar la posición que el lugar ocupa en el listado. Con el `id`, ya podemos actualizar la base de datos.

Como hemos visto, siempre que cambie algún contenido es importante que el adaptador actualice el `Cursor`, esto ya lo hace la función `guardar()`.

3. Para que los cambios en las fotografías se actualicen también has obtener el lugar de forma adecuada y llamar a `actualizaPosLugar()`:

```
public void ponerFoto(int pos, String uri, ImageView imageView) {
    Lugar lugar = lugares.elemento adaptador.lugarPosicion(pos);
    lugar.setFoto(uri);
    visualizarFoto(lugar, imageView);
    actualizaPosLugar(pos, lugar);
}
```

```
fun ponerFoto(pos: Int, uri: String?, imageView: ImageView) {
    val lugar = lugares.elemento adaptador.lugarPosicion(pos)
    lugar.foto = uri ?: ""
    visualizarFoto(lugar, imageView)
    actualizaPosLugar(pos, lugar)
}
```

4. Verifica que tanto los cambios de valoración como de fotografía se almacenan correctamente.



Ejercicio: Alta de un lugar

En este ejercicio aprenderemos a añadir nuevos registros a la base de datos.

1. Reemplaza en la clase `LugaresBD` el método `nuevo()` por el siguiente. Su finalidad es crear un nuevo lugar en blanco y devolver el `id` del nuevo lugar.

```
@Override public int nuevo() {
    int _id = -1;
    Lugar lugar = new Lugar();
    getWritableDatabase().execSQL("INSERT INTO lugares (nombre, " +
        "direccion, longitud, latitud, tipo, foto, telefono, url, " +
        "comentario, fecha, valoracion) VALUES ('', '', " +
        lugar.getPosicion().getLongitud() + ", "+
        lugar.getPosicion().getLatitud() + ", " + lugar.getTipo().ordinal() +
        ", '', 0, '', '', " + lugar.getFecha() + ", 0)");
    Cursor c = getReadableDatabase().rawQuery(
        "SELECT _id FROM lugares WHERE fecha = " + lugar.getFecha(), null);
    if (c.moveToNext()) _id = c.getInt(0);
    c.close();
    return _id;
}
```

```
override fun nuevo():Int {
    var _id = -1
    val lugar = Lugar(nombre = "")
    writableDatabase.execSQL("INSERT INTO lugares (nombre, direccion, " +
        "longitud, latitud, tipo, foto, telefono, url, comentario, " +
        "fecha, valoracion) VALUES ('', '', ${lugar.posicion.longitud}, " +
        "${lugar.posicion.latitud}, ${lugar.tipoLugar.ordinal}, '', 0, " +
        "''', '', ${lugar.fecha},0)")
    val c = readableDatabase.rawQuery((
        "SELECT _id FROM lugares WHERE fecha = " + lugar.fecha), null)
    if (c.moveToNext()) _id = c.getInt(0)
    c.close()
    return _id
}
```

Comenzamos inicializando el valor del `_id` a devolver a -1. De esta manera, si hay algún problema este será el valor devuelto. Luego se crea un nuevo objeto `Lugar`. Si consultas el constructor de la clase, observarás que solo se inicializan `posicion`, `tipo` y `fecha`. El resto de

los valores serán una cadena vacía para `String` y 0 para valores numéricos. Acto seguido, se crea una nueva fila con esta información. Los valores de texto y numéricos tampoco se indican, al inicializarse de la misma manera.

El método ha de devolver el `_id` del elemento añadido. Para conseguirlo se realiza una consulta buscando una fila con la misma fecha que acabamos de introducir.

2. Para la acción de añadir vamos a utilizar el botón flotante que tenemos desde la primera versión de la aplicación. Abre el fichero `res/layout/activity_main.xml` y reemplaza el icono aplicado a este botón:

```
<android.support.design.widget.FloatingActionButton
...
    android:src="@android:drawable/ic_input_add"
... />
```

3. Abre la clase `MainActivity` y dentro de `onCreate()` comenta el código tachado y añade el subrayado, para que se ejecute al pulsar el botón flotante:

```
fab.setOnClickListener(new View.OnClickListener() {
    @Override public void onClick(View view) {
        Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)
            .setAction("Action", null).show();
        usoLugar.nuevo();
    }
});
```

```
fab.setOnClickListener { view ->
    Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)
        .setAction("Action", null).show()
    usoLugar.nuevo()
}
```

4. Añade el siguiente caso de uso en `CasoUsoLugar`:

```
public void nuevo() {
    int id = lugares.nuevo();
    GeoPunto posicion = ((Aplicacion) actividad).getApplication()
        .posicionActual;
    if (!posicion.equals(GeoPunto.SIN_POSICION)) {
        Lugar lugar = lugares.elemento(id);
        lugar.setPosicion(posicion);
        lugares.actualiza(id, lugar);
    }
    Intent i = new Intent(actividad, EdicionLugarActivity.class);
    i.putExtra("_id", id);
    actividad.startActivity(i);
}
```

```
fun nuevo() {
    val _id = lugares.nuevo()
    val posicion = (actividad.application as Aplicacion).posicionActual
    if (posicion != GeoPunto.SIN_POSICION) {
        val lugar = lugares.elemento(_id)
        lugar.posicion = posicion
        lugares.actualiza(_id, lugar)
    }
    val i = Intent(actividad, EdicionLugarActivity::class.java)
    i.putExtra("_id", _id)
    actividad.startActivity(i)
}
```

Comenzamos creando un nuevo lugar en la base de datos cuyo identificador va a ser `_id`. La siguiente línea obtiene la posición actual. Si el dispositivo está localizado, obtenemos el lugar recién creado, cambiamos su posición y lo volvemos a guardar. A continuación, vamos a lanzar la actividad `EdicionLugarActivity` para que el usuario rellene los datos del lugar. Hasta ahora hemos utilizado el extra “pos” para indicar la posición en la lista del objeto a editar. Pero ahora esto no es posible, dado que este nuevo lugar no ha sido añadido a la lista. Para resolver el

problema vamos a crear un nuevo extra, “_id”, que usaremos para identificar el lugar a editar por medio de su campo _id.

5. En la clase `EdicionLugarActivity` añade el código subrayado:

```
private int _id;

@Override protected void onCreate(Bundle savedInstanceState) {
    ...
    Bundle extras = getIntent().getExtras();
    pos = extras.getInt("pos", -1);
    _id = extras.getInt("_id", -1);
    if ( _id!=-1) lugar = Lugares.elemento( _id);
    else lugar = adaptador.lugarPosicion(pos);
    actualizaVistas();
}
```

```
var _id = -1

override fun onCreate(savedInstanceState: Bundle?) {
    ...
    pos = intent.extras?.getInt("pos", -1) ?: -1
    _id = intent.extras?.getInt("_id", -1) ?: -1
    lugar = if ( _id != -1) Lugares.elemento( _id)
           else adaptador.lugarPosicion(pos)
    actualizaVistas()
}
```

Esta actividad va a poder ser llamada de dos formas alternativas: usando el extra “pos” para indicar que el lugar a modificar ha de extraerse de una posición del adaptador; o usando “_id” en este caso el lugar será extraído de la base de datos usando su identificador. Observa cómo se han definido dos variables globales, `pos` e `_id`. Aunque solo una se va a inicializar y la otra valdrá -1.

6. Cuando el usuario pulse la opción guardar se llamará al método `onOptionsItemSelected()`. Para almacenar la información tendremos que verificar cuál de las dos variables ha sido inicializada. Añade el código subrayado en este método:

```
case R.id.accion_guardar:
    ...
    if ( _id== -1) int _id = adaptador.idPosicion(pos);
    usoLugar.guardar(_id, lugar);
    finish();
    return true;
```

```
R.id.accion_guardar -> {
    ...
    if ( _id== -1) val _id = adaptador.idPosicion(pos)
    usoLugar.guardar(_id, nuevoLugar)
    finish()
    return true
}
```

El primer `if` es añadido dado que si nos han pasado el identificador `_id` ya no tiene sentido obtenerlo a partir de la posición.

7. Verifica que los cambios introducidos funcionan correctamente.



Ejercicio: Baja de un lugar

En este ejercicio aprenderemos a eliminar filas de la base de datos.

1. Reemplaza en la clase `LugaresBD` el método `borrar()` por el siguiente. Su finalidad es eliminar el lugar correspondiente al `id` indicado.

```
public void borrar(int id) {
```

```
getWritableDatabase().execSQL("DELETE FROM lugares WHERE _id = " + id);
}
```

```
override fun borrar(id: Int) {
    writableDatabase.execSQL("DELETE FROM lugares WHERE _id = $id")
}
```

2. Añade en la clase `VistaLugarActivity`, dentro del método `onOptionsItemSelected()`, el código subrayado:

```
case R.id.accion_borrar:
    int id = adaptador.idPosicion(pos);
    usoLugar.borrar(pos id)
    return true;
```

```
R.id.accion_borrar -> {
    val id = adaptador.idPosicion(pos)
    usoLugar.borrar(pos id)
    return true
}
```

3. En `CasosUsoLugares`, dentro de `borrarLugar()`, añade las dos líneas subrayadas para actualizar el cursor y notificar al adaptador que los datos han cambiado:

```
Lugares.borrar(id);
adaptador.setCursor(Lugares.extraeCursor());
adaptador.notifyDataSetChanged();
actividad.finish();
```

```
lugares.borrar(id)
adaptador.cursor = lugares.extraeCursor()
adaptador.notifyDataSetChanged()
actividad.finish()
```

4. Ejecuta la aplicación y trata de dar de baja algún lugar.



Práctica: Opción CANCELAR en el alta de un lugar

Si seleccionas la opción *nuevo* y en la actividad `EdicionLugarActivity` seleccionas la opción *CANCELAR*, puedes verificar que esta opción funciona mal. Los datos introducidos no se guardarán; sin embargo, se creará un nuevo lugar con todos sus datos en blanco. Para verificarlo has de salir de la aplicación para que se recargue el adaptador.

Para evitar este comportamiento, borra el elemento nuevo cuando se seleccione la opción *CANCELAR*. Pero este comportamiento ha de ser diferente cuando el usuario entró en la actividad `EdicionLugarActivity` para editar un lugar ya existente. Para diferenciar estas dos situaciones puedes utilizar los extras `_id` y `pos`.



Solución:

Añade en el método `onOptionsItemSelected()` en la opción `accion_cancelar`:

```
if (_id!=-1) Lugares.borrar(_id)
```



Preguntas de repaso: SQLite II

Loaders y LoaderManager

NOTA: Este apartado se plantea solo a modo de introducción. Abordar un estudio detallado en este curso de introducción resultaría excesivo.

La actualización de cursores provenientes de consultas en bases de datos es algo complejo. Además, tenemos una dificultad añadida: Android insiste en que estas consultas se realicen en segundo plano, para así no bloquear el hilo de la interfaz de usuario³.

Para resolver de forma sencilla este problema se introducen en Android 3.0 las clases **Loader** y **LoaderManager**. Ambas clases se encuentran en la librería de compatibilidad, por lo que pueden usarse desde la versión 1.6.

La gran ventaja de la clase **Loader** es que realiza todas las operaciones con el cursor de forma asíncrona, y por lo tanto, nunca bloquearemos el hilo de la interfaz de usuario. Además, cuando se gestiona por medio de un **LoaderManager**, un **Loader** conservará los datos de un cursor a través del ciclo de vida de una actividad (por ejemplo, cuando se reinicia debido a un cambio de configuración, o por falta de memoria). Como ventaja adicional, un **Loader** actúa de forma inteligente, controlando el origen de los datos para realizar las actualizaciones automáticamente cuando los datos cambian.

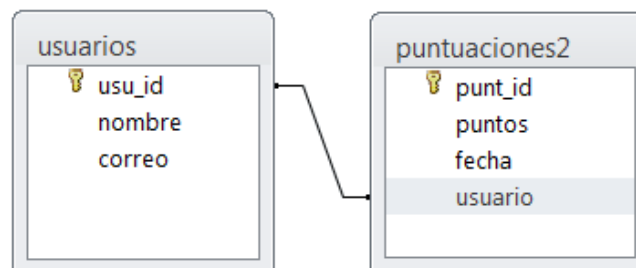
Más información sobre **Loader** en:

- <http://www.androiddesignpatterns.com/2012/07/loaders-and-loadermanager-background.html>
- <http://developer.android.com/guide/components/loaders.html>

9.7.4. Bases de datos relacionales

Una base de datos relacional es aquella que está formada por varias tablas, de forma que se han establecido relaciones o conexiones entre alguno de sus campos. Esto permite que cuando nos situemos en una fila de una tabla se acceda de forma automática a la fila de otra tabla a través de esta relación.

Veamos un ejemplo: imaginemos que queremos almacenar varios datos sobre los usuarios de nuestro juego: nombre, correo electrónico, fecha del último acceso, etc. También queremos seguir guardando la puntuación obtenida en cada partida por cada usuario. La mejor solución sería almacenar esta información en dos tablas diferentes y crear una relación entre ellas. A continuación, se muestra un esquema de la estructura a definir:



Cada una de estas dos tablas comienza con un identificador numérico: *usu_id* y *punt_id*, que serán utilizados como código de usuario y código de puntuación, respectivamente. En la tabla *puntuaciones2*, además de la información propia de una puntuación se ha añadido el campo *usuario*. En este campo se ha de almacenar el código de usuario que obtuvo la puntuación. Se ha establecido una relación entre este campo y *usu_id* de la tabla *usuarios*. Esta forma de trabajar resulta muy eficiente: además de ahorrar memoria, evita que se repliquen los datos.

puntuaciones2				usuarios		
punt_id	puntos	fecha	usuario	usu_id	nombre	correo
1	10000	27/9/12	2	1	Pepe	p@upv.es

³ <http://youtu.be/ekn9j2J9sos>

2	20000	28/9/12	2	→	2	Juan	j@upv.es
3	10000	28/9/12	1				



Ejercicio: Una base de datos relacional para las puntuaciones

Pasemos a demostrar cómo guardar las puntuaciones obtenidas en Asteroides en una base de datos relacional formada por dos tablas, tal y como se acaba de describir:

1. Crea la clase `AlmacenPuntuacionesSQLiteRel` en el proyecto `Asteroides`:

```
public class AlmacenPuntuacionesSQLiteRel extends SQLiteOpenHelper
    implements AlmacenPuntuaciones{
    public AlmacenPuntuacionesSQLiteRel(Context context) {
        super(context, "puntuaciones", null, 2);
    }

    //Métodos de SQLiteOpenHelper
    @Override public void onCreate(SQLiteDatabase db) {
        db.execSQL("CREATE TABLE usuarios (" +
            "usu_id INTEGER PRIMARY KEY AUTOINCREMENT, " +
            "nombre TEXT, correo TEXT)");
        db.execSQL("CREATE TABLE puntuaciones2 (" +
            "pun_id INTEGER PRIMARY KEY AUTOINCREMENT, " +
            "puntos INTEGER, fecha BIGINT, usuario INTEGER, " +
            "FOREIGN KEY (usuario) REFERENCES usuarios (usu_id))");
    }

    @Override public void onUpgrade(SQLiteDatabase db,
        int oldVersion, int newVersion) {
        // En el siguiente ejercicio se implementará este método
    }

    //Métodos de AlmacenPuntuaciones
    public List<String> listaPuntuaciones(int cantidad) {
        List<String> result = new ArrayList<String>();
        SQLiteDatabase db = getReadableDatabase();
        Cursor cursor = db.rawQuery("SELECT puntos, nombre FROM "
            + "puntuaciones2, usuarios WHERE usuario = usu_id ORDER BY "
            + "puntos DESC LIMIT " + cantidad, null);
        while (cursor.moveToNext()){
            result.add(cursor.getInt(0)+" " +cursor.getString(1));
        }
        cursor.close();
        db.close();
        return result;
    }

    public void guardarPuntuacion(int puntos, String nombre,
        long fecha) {
        SQLiteDatabase db = getWritableDatabase();
        guardarPuntuacion(db, puntos, nombre, fecha);
        db.close();
    }

    public void guardarPuntuacion(SQLiteDatabase db, int puntos,
        String nombre, long fecha) {
        int usuario = buscaInserta(db, nombre);
        db.execSQL("PRAGMA foreign_keys = ON");
        db.execSQL("INSERT INTO puntuaciones2 VALUES ( null, " +
            puntos + ", " + fecha + ", " + usuario + ")");
    }

    private int buscaInserta(SQLiteDatabase db, String nombre) {
        Cursor cursor = db.rawQuery("SELECT usu_id FROM usuarios "
            + "WHERE nombre='" + nombre + "'", null);
    }
}
```

```

    if (cursor.moveToNext()) {
        int result = cursor.getInt(0);
        cursor.close();
        return result;
    } else {
        cursor.close();
        db.execSQL("INSERT INTO usuarios VALUES (null, '" + nombre
            + "', 'correo@dominio.es')");
        return buscaInserta(db, nombre);
    }
}
}
}

```

El constructor de la clase se limita a llamar al constructor heredado de forma similar al ejemplo anterior. La diferencia es que ahora indicamos que es la versión 2 en lugar de la 1.

El método `onCreate()` se invoca solo cuando no existe la base de datos. En nuestro caso se crean las dos tablas y se establece la relación mediante el código SQL `"FOREIGN KEY (usuario) REFERENCES usuarios (usu_id)"`.

El método `listaPuntuaciones()` es similar a la versión anterior, aunque se ha modificado ligeramente la consulta SQL. Ahora, la cláusula `FROM` incluye las dos tablas, además, se ha añadido `WHERE usuario = usu_id` para asegurarnos que se cumpla la relación. El método `guardarPuntuacion()` ha sido sobrecargado para disponer de dos versiones. En el siguiente ejercicio se clarificará por qué se ha actuado así. En la segunda versión se comienza llamando a `buscaInserta()` para obtener el código de usuario. Con esta información se crea una nueva entrada en la tabla `puntuaciones2`. Observa como antes se ha añadido el comando SQL `PRAGMA foreign_keys = ON`. Hay que ejecutarlo cada vez que abramos la base de datos para que el sistema realice una verificación automática de las relaciones definidas. Es decir, si se crea una nueva puntuación con un usuario que no existe en la tabla `usuarios` o se borra un usuario con puntuaciones; se producirá una excepción.

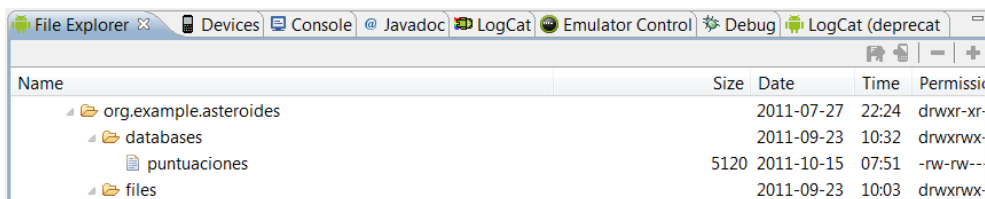
NOTA: Recuerda ejecutar este comando cada vez que abras la base de datos.

Finalmente tenemos el método `buscaInserta()`. Comienza buscando el nombre de usuario. Si existe, devolverá su código; en caso contrario se ejecuta el comando SQL necesario para crear un nuevo usuario con este nombre. Tenemos que devolver el código del nuevo usuario. Esta información se obtiene realizando una llamada recursiva.

2. Abre el fichero `MainActivity.java` y modifica el método `onCreate()` para que se pueda ejecutar la siguiente línea:

```
almacen = new AlmacenPuntuacionesSQLiteRel(this);
```

3. Abre la vista `File Explorer` y busca la ruta `/data/data/org.example.asteroides`.
4. Elimina los ficheros de la carpeta `databases`.



Name	Size	Date	Time	Permissions
org.example.asteroides		2011-07-27	22:24	drwxr-xr-
databases		2011-09-23	10:32	drwxrwx--
puntuaciones	5120	2011-10-15	07:51	-rw-rw---
files		2011-09-23	10:03	drwxrwx--

De no hacerlo no se llamaría al método `onCreate()`, al existir ya una base de datos. Esto ocasionaría un error, al no coincidir las tablas usadas en esta nueva versión con las de la base de datos. Una forma alternativa de eliminar las bases de datos es utilizar el administrador de aplicaciones del terminal, buscar la aplicación Asteroides y usar la opción `Borrar datos`. En el siguiente ejercicio veremos cómo evitar la necesidad de hacer esta tarea.

5. Verifica que las puntuaciones se almacenan correctamente.

9.7.5. El método *onUpgrade* de la clase *SQLiteOpenHelper*

El sistema Android facilita la actualización de las aplicaciones con una intervención mínima por parte del usuario. En este contexto, resulta muy importante que todos los datos introducidos por el usuario no se pierdan en una actualización.

Una nueva versión de nuestra aplicación suele requerir algún cambio en las estructuras de datos. Como hemos visto, la clase *SQLiteOpenHelper* dispone del método *onUpgrade* previsto con esta finalidad. El siguiente ejercicio nos muestra cómo utilizarlo.



Ejercicio: Utilizando el método *onUpgrade* de la clase *SQLiteOpenHelper*

1. Elimina el fichero con las bases de datos como se ha indicado en el punto 4 del ejercicio anterior.
2. Ejecuta Asteroides y selecciona la clase *AlmacenPuntuacionesSQLite* para el almacenamiento de datos. Sal de Asteroides.
3. Ejecuta Asteroides y destruye algún asteroide para obtener una puntuación. Verifica que se ha vuelto a crear el fichero de base de datos.
4. Abre la clase *AlmacenPuntuacionesSQLiteRel* y reemplaza el siguiente método:

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    if (oldVersion==1 && newVersion==2){
        onCreate(db); //Crea las nuevas tablas
        Cursor cursor = db.rawQuery("SELECT puntos, nombre, fecha "+
            "FROM puntuaciones",null); //Recorre la tabla antigua
        while (cursor.moveToNext()) {
            guardarPuntuacion(db, cursor.getInt(0), cursor.getString(1),
                cursor.getInt(2)); //Crea los nuevos registros
        }
        cursor.close();
        db.execSQL("DROP TABLE puntuaciones"); //Elimina tabla antigua
    }
}
```

Este código se ejecutará solo una vez, cuando el sistema detecta que se está instalando la versión 2 de la base de datos, pero se encuentra ya instalada la versión 1. Básicamente se crean las nuevas tablas (*onCreate(db)*), se recorre la tabla antigua para transportar la información a las nuevas y se termina borrando la tabla antigua. Recuerda que el número de versión se indica en el constructor.

5. Ejecuta Asteroides y selecciona la clase *AlmacenPuntuacionesSQLiteRel* para el almacenamiento de datos. Sal de Asteroides.
6. Ejecuta Asteroides (mejor si lo haces en modo *Debug* poniendo un *break point* al principio de este método) y muestra la lista de puntuaciones. Han de aparecer las mismas puntuaciones.

9.8. Content Provider

Los proveedores de contenido (*ContentProviders*) son uno de los bloques constructivos más importantes de Android. Nos van a permitir acceder a información proporcionada por otras aplicaciones, o a la inversa, compartir nuestra información con otras aplicaciones.

Tras describir los principios en los que se basan los *ContentProviders*, pasaremos a demostrar cómo acceder a *ContentProviders* creados por otras aplicaciones. Esta operación resulta muy útil, dado que te permitirá tener acceso a información interesante, como la lista de contactos, el registro de llamadas o los ficheros multimedia almacenados en el dispositivo. Terminaremos este apartado mostrando cómo crear tu propio *ContentProvider*, de forma que otras aplicaciones puedan acceder a tu información.

9.8.1. Conceptos básicos

El modelo de datos

La forma en la que un ContentProvider almacena la información es un aspecto de diseño interno, de forma que podríamos utilizar cualquiera de los métodos descritos en este capítulo. No obstante, cuando hagamos una consulta al ContentProvider se devolverá un objeto de tipo `Cursor`. Esto hace que la forma más práctica para almacenar la información sea en una base de datos.

Utilizando términos del modelo de bases de datos, un ContentProvider proporciona sus datos a través de una tabla simple, donde cada fila es un registro y cada columna es un tipo de datos con un significado particular. Por ejemplo, el ContentProvider que utilizaremos en el siguiente ejemplo se llama `CallLog`, y permite acceder al registro de llamadas del teléfono. La información que nos proporciona tiene la estructura que se muestra a continuación:

<code>_id</code>	<code>Date</code>	<code>Number</code>	<code>Duration</code>	<code>Type</code>
1	12/10/10 16:10	555123123	65	INCOMING_TYPE
3	12/11/10 20:42	555453455	356	OUTGOING_TYPE
4	13/11/10 12:15	555123123	90	MISSED_TYPE
5	14/11/10 22:10	555783678	542	OUTGOING_TYPE

Tabla 2: Ejemplo de estructura de datos de un ContentProvider.

Cada registro incluye el campo numérico `_id` que lo identifica de forma única. Como veremos a continuación, podremos utilizar este campo para identificar una llamada en concreto.

La forma de acceder a la información de un ContentProvider es muy similar al proceso descrito para las bases de datos. Es decir, vamos a poder realizar una consulta (incluso utilizando el lenguaje SQL), tras la cual se nos proporcionará un objeto de tipo `Cursor`. Este objeto contiene una información con una estructura similar a la mostrada en la tabla anterior.

Las URI

Para acceder a un ContentProvider en particular, será necesario identificarlo con una URI. Una URI (véase estándar RFC 2396) es una cadena de texto que permite identificar un recurso de información. Suele utilizarse frecuentemente en Internet (por ejemplo, para acceder a una página web). Una URI está formada por cuatro partes, tal y como se muestra a continuación:

```
<standard_prefix>://<authority>/<data_path>/<id>
```

La parte `<standard_prefix>` de todos los proveedores de contenido ha de ser siempre `content`. En el primer ejemplo de este apartado utilizaremos un ContentProvider donde Android almacena el registro de llamadas. Para acceder a este ContentProvider utilizaremos la siguiente URI:

```
content://call_log/calls
```

En la Tabla 12 encontrarás otros ejemplos de URI que te permitirán acceder a otras informaciones almacenadas en Android.

Para acceder a un elemento concreto has de indicar un `<id>` en la URI. Por ejemplo, si te interesa solo acceder a la llamada con identificador 4 (normalmente corresponderá a la cuarta llamada de la lista), has de indicar:

```
content://call_log/calls/4
```

Un mismo ContentProvider puede contener múltiples conjuntos de datos identificados por diferentes URI. Por ejemplo, para acceder a los ficheros multimedia almacenados en el móvil utilizaremos el ContentProvider `MediaStore`, utilizando algunos de los siguientes ejemplos de URI:

```
content://media/internal/images
content://media/external/video/5
content://media/*/audio
```


Cada `ContentProvider` suele definir constantes de `String` con sus correspondientes URI. Por ejemplo, `android.provider.CallLog.Calls.CONTENT_URI` corresponde a `"content://call_log/calls"`. Y la constante `android.provider.MediaStore.Audio.Media.INTERNAL_CONTENT_URI` corresponde a `"content://media/internal/audio"`.

9.8.2. Acceder a la información de un `ContentProvider`

Android utiliza los proveedores de contenido para almacenar diferentes tipos de información. Veamos los más importantes en la tabla siguiente:

Clase	Información almacenada	Ejemplos de URI
Browser	Enlaces favoritos, historial de navegación, historial de búsquedas.	<code>content://browser/bookmarks</code>
CallLog	Llamadas entrantes, salientes y perdidas.	<code>content://call_log/calls</code>
Contacts	Lista de contactos del usuario.	<code>content://contacts/people</code>
MediaStore	Ficheros de audio, vídeo e imágenes, almacenados en dispositivos de almacenamiento internos y externos.	<code>content://media/internal/images</code> <code>content://media/external/video</code> <code>content://media/*/audio</code>
Setting	Preferencias del sistema.	<code>content://settings/system/ringtone</code> <code>content://settings/system/notification_sound</code>
UserDictionary (a partir de 1.5)	Palabras definidas por el usuario, utilizadas en los métodos de entrada predictivos.	<code>content://user_dictionary/words</code>
Telephony (a partir de 1.5)	Mensajes SMS y MMS mandados o recibidos desde el teléfono.	<code>content://sms</code> <code>content://sms/inbox</code> <code>content://sms/sent</code> <code>content://mms</code>
Calendar (a partir de 4.0)	Permite consultar y editar los eventos del calendario.	<code>content://com.android.calendar/time</code> <code>content://com.android.calendar/events</code>
Document (a partir de 4.4)	Permite acceder a ficheros locales o en la nube.	<code>content://com.dropbox.android.Dropbox/metadata/</code> <code>content://com.dominio.mio/dir/fich.txt</code>

Tabla 3: *ContentProviders disponibles en Android.*

Leer información de un `ContentProvider`

Veamos un ejemplo que permite leer el registro de llamadas del teléfono. Crea una nueva aplicación y llámala `ContentCallLog`.

Reemplaza el contenido del fichero `res/layout/activity_main.xml` por:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:id="@+id/salida"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>
</LinearLayout>
```

De esta forma podremos identificar el `TextView` desde el programa y utilizarlo para mostrar la salida.

Añade en `AndroidManifest.xml` las líneas:

```
<uses-permission
```

```
android:name="android.permission.READ_CALL_LOG"/>
```

Al solicitar el permiso `READ_CALL_LOG` podremos acceder al registro de llamadas. Añade al final del método `onCreate()` de la actividad principal el siguiente código:

```
...
String[] TIPO_LLAMADA = {"", "entrante", "saliente", "perdida",
    "mensaje de voz", "cancelada", "lista bloqueados"};
TextView salida = findViewById(R.id.salida);
Uri llamadas = Uri.parse("content://call_log/calls");
Cursor c = getContentResolver().query(llamadas, null, null, null, null);
while (c.moveToNext()) {
    salida.append("\n" + DateFormat.format("dd/MM/yy k:mm (",
        c.getLong(c.getColumnIndex(Calls.DATE)))
        + c.getString(c.getColumnIndex(Calls.DURATION)) + ") "
        + c.getString(c.getColumnIndex(Calls.NUMBER)) + ", "
        + TIPO_LLAMADA[Integer.parseInt(c.getString(
            c.getColumnIndex(Calls.TYPE)))]);
}
```

En primer lugar se define un *array* de *strings*, de forma que `TIPO_LLAMADA[1]` corresponda a “entrante”, `TIPO_LLAMADA[2]` corresponda a “saliente”, etc. Luego se crea el objeto `salida`, que es asignado al `TextView` correspondiente del *layout*.

Ahora comienza lo interesante: creamos la URI `llamadas` asociada a `content://call_log/calls`. Para realizar la consulta creamos el `Cursor`, `c`. Se trata de la misma clase que hemos utilizado para hacer consultas en una base de datos. A través de `getContentResolver()`, obtenemos un `ContentResolver` asociado a la actividad, con el que podemos llamar al método `query()`. Este método permite varios parámetros para indicar exactamente los elementos que nos interesan, de forma similar a como se hace en una base de datos. Estos parámetros se estudiarán más adelante. Al no indicar ninguno se devolverán todas las llamadas registradas.

No tenemos más que desplazarnos por todos los elementos del cursor (`c.moveToNext()`) e ir añadiendo en la salida (`salida.append()`) la información correspondiente a cada registro. En concreto, fecha, duración, número de teléfono y tipo de llamada. Una vez que el cursor se encuentra situado en una fila determinada, podemos obtener la información de una columna utilizando los métodos `getString()`, `getInt()`, `getLong()` y `getFloat()`, dependiendo del tipo de dato almacenado. Estos métodos necesitan como parámetros el índice de columna. Para averiguar el índice de cada columna utilizaremos el método `getColumnIndex()`, indicando el nombre de la columna. En nuestro caso, estos nombres son “date”, “duration”, “number” y “type”. En el ejemplo, en lugar de utilizar estos nombres directamente se han utilizado las cuatro constantes con estos valores en la clase `Calls`.

Especial atención merece la columna “date”, que nos devuelve un entero largo que representa un instante concreto de una fecha. Para mostrarla en el formato deseado hemos utilizado el método estático `format()` de la clase `DateFormat`.

A continuación, se muestra el resultado de ejecutar este programa:

```
ContentCallLog
Hello World, ContentCallLog!
14/07/10 11:28 (455) 679314350, entrante
14/07/10 11:50 (84) 635478108, entrante
14/07/10 19:33 (0) 618614205, perdida
14/07/10 20:38 (0) 618614205, saliente
14/07/10 20:38 (30) 679314350, saliente
14/07/10 20:44 (35) 618614205, entrante
15/07/10 22:38 (371) 657638814, entrante
16/07/10 14:40 (66) 657638814, entrante
```

Veamos con más detalle el método `query()`:

```
Cursor query(Uri uri, String[] proyeccion, String seleccion, String[]
    argsSelecc, String orden)
```

Donde los parámetros corresponden a:

<code>uri</code>	URI correspondiente al ContentProvider a consultar.
<code>proyeccion</code>	Lista de columnas que queremos que nos devuelva.
<code>seleccion</code>	Cláusula SQL correspondiente a <code>WHERE</code> .
<code>argsSelecc</code>	Lista de argumentos utilizados en el parámetro <code>seleccion</code> .
<code>orden</code>	Cláusula SQL correspondiente a <code>ORDER BY</code> .

Para ilustrar el uso de estos parámetros, reemplaza en el ejemplo anterior:

```
Cursor c = getContentResolver.query(llamadas, null, null, null, null);
```

por:

```
String[] proyeccion = new String[] {
    Calls.DATE, Calls.DURATION, Calls.NUMBER, Calls.TYPE };
String[] argsSelecc = new String[] {"1"};
Cursor c = getContentResolver.query(
    llamadas,        // Uri del ContentProvider
    proyeccion,       // Columnas que nos interesan
    "type = ?",       // consulta WHERE
    argsSelecc,       // parámetros de la consulta anterior
    "date DESC");    // Ordenado por fecha, orden descendiente
```

De esta forma nos devolverán solo las columnas indicadas en `proyeccion`. Esto supone un ahorro de memoria en caso de que existan muchas columnas. En el siguiente parámetro se indican las filas que nos interesan por medio de una consulta de tipo `WHERE`. En caso de encontrar algún carácter `?`, este es sustituido por el primer `string` del parámetro `argSelecc`. En caso de haber más caracteres `?` se irían sustituyendo siguiendo el mismo orden. Cuando se sustituyen los interrogantes, cada elemento de `argSelecc` es introducido entre comillas. Por lo tanto, en el ejemplo la consulta `WHERE` resultante es `"WHERE type = '1'"`. Esta consulta implica que solo se mostrarán las llamadas entrantes. El último parámetro sería equivalente a indicar en SQL `"SORTED BY date DESC"`; es decir, el resultado estará ordenado por fecha en orden descendiente.

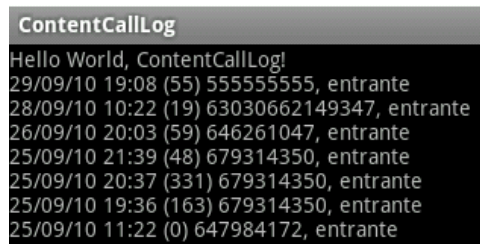
Escribir información en un ContentProvider

Añadir un nuevo elemento en un ContentProvider resulta muy sencillo. Para ilustrar cómo se hace, escribe el siguiente código al principio del ejemplo anterior:

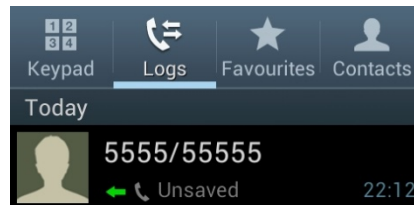
```
ContentValues valores = new ContentValues();
valores.put(Calls.DATE, new Date().getTime() );
valores.put(Calls.NUMBER, "555555555");
valores.put(Calls.DURATION, "55");
valores.put(Calls.TYPE, Calls.INCOMING_TYPE);
Uri nuevoElemento = getContentResolver().insert(
    Calls.CONTENT_URI, valores);
```

NOTA: Has de utilizar el `import java.util.Date`.

Como puedes ver, comenzamos creando un objeto `ContentValues`, donde vamos almacenado una serie de pares de valores, nombre de la columna y valor asociado a la columna. A continuación, se llama a `getContentResolver().insert()` y se le pasa la URI del ContentProvider y los valores a insertar. Este método nos devuelve una URI que apunta de forma específica al elemento que acabamos de insertar. Podrías utilizar esta URI para hacer una consulta y obtener un cursor al nuevo elemento y así poder modificarlo, borrarlo u obtener el `_ID`. Recuerda que has de pedir el permiso `WRITE_CALL_LOG`. Si ejecutas ahora el programa, la nueva llamada insertada ha de aparecer en primer lugar.



Estamos modificando el registro de llamadas del sistema; por lo tanto, también puedes verificar esta información desde las aplicaciones del sistema.



Borrar y modificar elementos de un ContentProvider

Puedes utilizar el método `delete()` para eliminar elementos de un ContentProvider:

```
int ContentProvider.delete(Uri uri, String seleccion, String[] argsSelecc)
```

Este método devuelve el número de elementos eliminados. Los tres parámetros del método se detallan a continuación:

- `uri` URI correspondiente al ContentProvider a consultar.
- `seleccion` Cláusula SQL correspondiente a `WHERE`.
- `argsSelecc` Lista de argumentos utilizados en el parámetro `seleccion`.

Si quisiéramos eliminar un solo elemento, podríamos obtener su URI e indicarlo en el primer parámetro, dejando los otros dos a `null`. Si por el contrario quieres eliminar varios elementos, puedes utilizar el parámetro `seleccion`. Por ejemplo, si quisiéramos eliminar todos los registros de llamada del número 555555555, escribiríamos:

```
getContentResolver().delete(Calls.CONTENT_URI, "number='555555555'", null)
```

También puedes utilizar el método `update()` para modificar elementos de un ContentProvider:

```
int ContentProvider.update(Uri uri, ContentValues valores,
    String seleccion, String[] argsSelecc)
```

Por ejemplo, si quisiéramos modificar los registros con número 555555555 por el número 444444444, escribiríamos:

```
ContentValues valores2 = new ContentValues();
valores2.put(Calls.NUMBER, "444444444");
getContentResolver().update(Calls.CONTENT_URI, valores2,
    "number='555555555'", null);
```

9.8.3. Creación de un ContentProvider

NOTA: Se trata de un aspecto avanzado no necesario en la mayoría de aplicaciones.

En este apartado vamos a describir los pasos necesarios para crear tu propio ContentProvider. En concreto, vamos a seguir tres pasos:

1. Definir una estructura de almacenamiento para los datos. En nuestro caso usaremos una base de datos SQLite.
2. Crear nuestra clase extendiendo `ContentProvider`.

3. Declarar el ContentProvider en el `AndroidManifest.xml` de nuestra aplicación.

Si no deseas compartir tu información con otras aplicaciones, no es necesario crear un ContentProvider. En ese caso resulta mucho más sencillo usar una base de datos directamente, tal y como se ha explicado en el apartado anterior. En este apartado vamos a seguir el mismo ejemplo descrito en los anteriores apartados del capítulo, es decir, vamos a crear un ContentProvider que nos permita compartir la lista de puntuaciones con otras aplicaciones. Posiblemente, no se trate de una situación muy realista. Es de suponer que ninguna aplicación estará interesada en esta información. No obstante, por razones didácticas resulta más sencillo continuar con el mismo ejemplo.

Crea una nueva aplicación que se llame PuntuacionesProvider y cuyo nombre de paquete sea `org.example.puntuacionesprovider`.

Definir la estructura de almacenamiento del ContentProvider

El objetivo último de un ContentProvider es almacenar información de forma permanente. Por lo tanto, resulta imprescindible utilizar alguno de los mecanismos descritos en este tema, o en el siguiente, para almacenar datos. Como se ha estudiado en el apartado anterior, podemos realizar consultas a un ContentProvider de forma similar a una base de datos (podemos hacer consultas SQL y nos devuelve un objeto de tipo `Cursor`). Por lo tanto, la forma más sencilla de almacenar los datos de un ContentProvider es en una base de datos. De esta forma, si nos solicitan una consulta SQL, no tendremos más que trasladarla a nuestra base de datos, y el objeto `Cursor` que nos devuelva será el resultado que nosotros devolveremos.

Para crear la base de datos de nuestro ContentProvider, añade una nueva clase que se llame `PuntuacionesSQLiteHelper` e introduce el siguiente código:

```
public class PuntuacionesSQLiteHelper extends SQLiteOpenHelper {

    public PuntuacionesSQLiteHelper(Context context) {
        super(context, "puntuaciones", null, 1);
    }

    @Override public void onCreate(SQLiteDatabase db) {
        db.execSQL("CREATE TABLE puntuaciones ("
            + "_id INTEGER PRIMARY KEY AUTOINCREMENT, "
            + "puntos INTEGER, "
            + "nombre TEXT, "
            + "fecha BIGINT)");
    }

    @Override public void onUpgrade(SQLiteDatabase db, int oldVersion,
                                    int newVersion) {
        // En caso de una nueva versión habría que actualizar las tablas
    }
}
```

La clase que acabamos de añadir al proyecto se encarga de crear la base de datos `puntuaciones` extendiendo la clase `SQLiteOpenHelper`. Este proceso se ha descrito en el apartado dedicado a las bases de datos. Dado que estamos trabajando sobre el mismo ejemplo, la tabla creada es idéntica y lo mismo ocurre con el código. Para una explicación más detallada recomendamos consultar dicho apartado.

Extendiendo la clase ContentProvider

Ahora abordamos la parte más laboriosa: la creación de una clase descendiente de `ContentProvider`.

Los métodos principales que tenemos que implementar son:

- `getType()` – devuelve el tipo MIME de los elementos del ContentProvider.
- `query()` – permite realizar consultas al ContentProvider.
- `insert()` – inserta nuevos datos.

`delete()` – borra elementos del `ContentProvider`.

`update()` – permite modificar los datos existentes.

La clase `ContentProvider` es *thread-safe*, es decir, toma las precauciones necesarias para evitar problemas con las llamadas simultáneas de varios procesos. Por lo tanto, en la creación de una subclase no nos tenemos que preocupar de este aspecto.

Añade una nueva clase que se llame `PuntuacionesProvider` e introduce el siguiente código:

```
public class PuntuacionesProvider extends ContentProvider {
    public static final String AUTORIDAD =
        "org.example.puntuacionesprovider";
    public static final Uri CONTENT_URI = Uri.parse("content://"
        + AUTORIDAD + "/puntuaciones");
    public static final int TODOS_LOS_ELEMENTOS = 1;
    public static final int UN_ELEMENTO = 2;
    private static UriMatcher URI_MATCHER = null;
    static {
        URI_MATCHER = new UriMatcher(UriMatcher.NO_MATCH);
        URI_MATCHER.addURI(AUTORIDAD, "puntuaciones", TODOS_LOS_ELEMENTOS);
        URI_MATCHER.addURI(AUTORIDAD, "puntuaciones/#", UN_ELEMENTO);
    }
    public static final String TABLA = "puntuaciones";
    private SQLiteDatabase baseDeDatos;

    @Override public boolean onCreate() {
        PuntuacionesSQLiteHelper dbHelper = new
            PuntuacionesSQLiteHelper(getContext());
        baseDeDatos = dbHelper.getWritableDatabase();
        return baseDeDatos != null && baseDeDatos.isOpen();
    }
}
```

Como no podría ser de otra forma, la clase extiende `ContentProvider`. A continuación creamos constantes, `AUTORIDAD` y `CONTENT_URI`, que identificarán nuestro `ContentProvider` mediante la URI:

`content://org.example.puntuacionesprovider/puntuaciones`

Las siguientes líneas permiten crear el objeto estático `URI_MATCHER` de la clase `UriMatcher`. Es habitual en Java utilizar variables estáticas finales para albergar objetos que no van a cambiar en toda la vida de la aplicación, es decir, que son constantes. Conviene recordar que solo se creará un objeto `URI_MATCHER` aunque se instancie varias veces la clase `PuntuacionesProvider`. La clase `UriMatcher` permite diferenciar entre diferentes tipos de URI que vamos a manipular. En nuestro caso, permitimos dos tipos de URI: acabada en `/puntuaciones`, que identifica todas las puntuaciones almacenadas, y acabada en `/puntuaciones/#`, donde hay que reemplazar `#` por un código numérico que coincida con el campo `_id` de nuestra tabla. Cada tipo de URI ha de tener un código numérico asociado, en nuestro caso `NO_MATCH` (-1), `TODOS_LOS_ELEMENTOS` (1) y `UN_ELEMENTO` (2).

La declaración de variables termina con la constante `TABLA`, que identifica la tabla que gastaremos para almacenar la información y el objeto `baseDeDatos` donde se almacenará la información.

A continuación está el método `onCreate()`, que se llama cuando se crea una instancia de esta clase. Básicamente se crea un `SQLiteHelper` a partir de la clase descrita en el apartado anterior (`PuntuacionesSQLiteHelper`) y se asigna la base de datos resultante a la variable `baseDeDatos`. Devolvemos `true` solo en el caso de que no haya habido problemas en su creación.

Veamos la implementación del método `getType()`, que a partir de una URI nos devuelve el tipo MIME que le corresponde:

```
@Override public String getType(final Uri uri) {
    switch (URI_MATCHER.match(uri)) {
        case TODOS_LOS_ELEMENTOS:
            return "vnd.android.cursor.dir/vnd.org.example.puntuacion";
        case UN_ELEMENTO:
            return "vnd.android.cursor.item/vnd.org.example.puntuacion";
    }
}
```



```

        return "vnd.android.cursor.item/vnd.org.example.puntuacion";
    default:
        throw new IllegalArgumentException("URI incorrecta: " + uri);
    }
}

```

NOTA: Los tipo MIME (Multipurpose Internet Mail Extensions) fueron creados para identificar un tipo de datos concreto que añadíamos como anexo a un correo electrónico, aunque en la actualidad son utilizados por muchos sistemas y protocolos. Un tipo MIME tiene dos partes: *tipo_genérico/tipo_específico*; por ejemplo, *image/gif*, *image/jpeg*, *text/html*, etc.

Existe un convenio para identificar los tipos MIME que proporciona un ContentProvider. Si se trata de un recurso único, utilizamos:

`vnd.android.cursor.item/vnd.ELEMENTO`

Y si se trata de una colección de recursos, utilizamos:

`vnd.android.cursor.dir/vnd.ELEMENTO`

Donde `ELEMENTO` ha de ser reemplazado por un identificador que describa el tipo de datos. En nuestro caso hemos elegido `org.example.puntuacion`. Utilizar prefijos para definir el elemento minimiza el riesgo de confusión con otro ya existente.

A continuación hemos de sobrescribir los métodos `query`, `insert`, `delete` y `update`, que permitan consultar, insertar, borrar y actualizar elementos de nuestro ContentProvider. Comencemos por el primer método:

```

@Override public Cursor query(Uri uri, String[] proyeccion, String
                             seleccion, String[] argSeleccion, String orden) {
    SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();
    queryBuilder.setTables(TABLA);
    switch (URI_MATCHER.match(uri)) {
        case TODOS_LOS_ELEMENTOS:
            break;
        case UN_ELEMENTO:
            String id = uri.getPathSegments().get(1);
            queryBuilder.appendWhere("_id = " + id);
            break;
        default:
            throw new IllegalArgumentException("URI incorrecta "+uri);
    }
    return queryBuilder.query(baseDeDatos, proyeccion, seleccion,
                              argSeleccion, null, null, orden);
}

```

El método `query()` que hemos de implementar tiene parámetros similares a `SQLiteQueryBuilder.query()`. Por lo tanto, no tenemos más que trasladar la consulta a nuestra base de datos. No obstante, existe un pequeño inconveniente si nos pasan una URI que identifique un solo elemento, como la mostrada a continuación:

`content://org.example.puntuacionesprovider/puntuaciones/324`

Hemos de asegurarnos que solo se devuelve el elemento con `_id = 324`. Para conseguirlo se introduce una cláusula `switch`, que en caso de tratarse de una URI de tipo `UN_ELEMENTO`, añade a la cláusula `WHERE` de la consulta la condición correspondiente mediante el método `appendWhere()`. La cláusula `WHERE` puede tener más condiciones si se ha utilizado el parámetro `seleccion`.

```

@Override public Uri insert(Uri uri, ContentValues valores) {
    long IdFila = baseDeDatos.insert(TABLA, null, valores);
    if (IdFila > 0) {
        return ContentUris.withAppendedId(CONTENT_URI, IdFila);
    } else {
        throw new SQLException("Error al insertar registro en "+uri);
    }
}
@Override

```



```

public int delete(Uri uri, String seleccion, String[] argSeleccion) {
    switch (URI_MATCHER.match(uri)) {
        case TODOS_LOS_ELEMENTOS:
            break;
        case UN_ELEMENTO:
            String id = uri.getPathSegments().get(1);
            if (TextUtils.isEmpty(seleccion)) {
                seleccion = "_id = " + id;
            } else {
                seleccion = "_id = " + id + " AND (" + seleccion + ")";
            }
            break;
        default:
            throw new IllegalArgumentException("URI incorrecta: " + uri);
    }
    return baseDeDatos.delete(TABLA, seleccion, argSeleccion);
}

```

El método `insert()` no requiere explicaciones adicionales.

El método `delete()`, igual como ocurrió con el método `query()`, presenta el inconveniente de que pueden habernos indicado una URI que identifique un solo elemento (`.../puntuaciones/324`). En el método `query()` solucionamos este problema llamando a `SQLiteQueryBuilder.appendWhere()`. Sin embargo, ahora no disponemos de un objeto de esta clase, por lo que nos vemos obligados a realizar este trabajo a mano. En caso de no haberse indicado nada en `seleccion`, este parámetro valdrá `"_id = 324"`; y en caso de haberse introducido una condición, por ejemplo `"numero = '555'"`, `seleccion`, valdrá `"_id = 324 AND (numero = '555')"`.

```

@Override public int update(Uri uri, ContentValues valores, String
                        seleccion, String[] ArgumentosSeleccion) {
    switch (URI_MATCHER.match(uri)) {
        case TODOS_LOS_ELEMENTOS:
            break;
        case UN_ELEMENTO:
            String id = uri.getPathSegments().get(1);
            if (TextUtils.isEmpty(seleccion)) {
                seleccion = "_id = " + id;
            } else {
                seleccion = "_id = " + id + " AND (" + seleccion + ")";
            }
            break;
        default:
            throw new IllegalArgumentException("URI incorrecta: " + uri);
    }
    return baseDeDatos.update(TABLA, valores, seleccion,
                                ArgumentosSeleccion);
} } // fin de la clase

```

Finalizamos con el método `update()`, que es muy similar a `delete()`.

Declarar el ContentProvider en *AndroidManifest.xml*

Si queremos que nuestro `ContentProvider` sea visible para otras aplicaciones, resulta imprescindible hacérselo saber al sistema declarándolo en el `AndroidManifest.xml`. Para conseguirlo no tienes más que añadir el siguiente código dentro de la etiqueta `<application>`:

```

<provider
    android:authorities="org.example.puntuacionesprovider"
    android:name="org.example.puntuacionesprovider.PuntuacionesProvider"
    android:exported="true"/>

```

La declaración de un `ContentProvider` requiere que se especifiquen los atributos:

name: nombre cualificado de la clase donde hemos implementado nuestro `ContentProvider`.

authorities: parte correspondiente a la autoridad de las URI que vamos a publicar. Puede indicarse más de una autoridad.

También se pueden indicar otros atributos en la etiqueta `<provider>`. Veamos los más importantes:

label: etiqueta que describe el ContentProvider que se mostrará al usuario. Es una referencia a un recurso de tipo *string*.

icon: una referencia a un recurso de tipo *drawable* con un icono que represente nuestro ContentProvider.

enabled: indica si está habilitado. El valor por defecto es **true**.

exported: indica si se puede acceder a él desde otras aplicaciones. El valor por defecto es **false**. Si está en **false** solo podrá ser utilizado por aplicaciones con el mismo *id* de usuario que la aplicación donde se crea.

readPermission: permiso requerido para consultar el ContentProvider.

writePermission: permiso requerido para modificar el ContentProvider.

permission: permiso requerido para consultar o modificar el ContentProvider. Sin efecto si se indica **readPermission** o **writePermission**. Para más información, consúltase el capítulo sobre seguridad.

multiprocess: indica si cualquier proceso puede crear una instancia del ContentProvider (**true**) o solo el proceso de la aplicación donde se ha creado (**false**, valor por defecto).

process: nombre del proceso en el que el ContentProvider ha de ejecutarse. Habitualmente, todos los componentes de una aplicación se ejecutan en el mismo proceso creado para la aplicación. Si no se indica lo contrario, el nombre del proceso coincide con el nombre del paquete de la aplicación (si lo deseas puedes cambiar este nombre con el atributo **process** de la etiqueta `<application>`). Si prefieres que un componente de la aplicación se ejecute en su propio proceso, has de utilizar este atributo.

initOrder: orden en que el ContentProvider ha de ser instalado en relación con otros ContentProvider del mismo proceso.

syncable: indica si la información del ContentProvider está sincronizada con un servidor.

9.8.4. Acceso a PuntuacionesProvider desde Asteroides

Una vez hemos creado y declarado nuestro ContentProvider, vamos a probarlo desde la aplicación Asteroides. Como hemos hecho en ejemplos anteriores, vamos a crear una nueva clase que implemente la interfaz **AlmacenPuntuaciones**.

Crea una nueva clase en la aplicación **Asteroides** con el nombre **AlmacenPuntuacionesProvider**. Introduce el siguiente código:

```
public class AlmacenPuntuacionesProvider implements AlmacenPuntuaciones {
    private Activity activity;

    public AlmacenPuntuacionesProvider(Activity activity) {
        this.activity = activity;
    }

    public void guardarPuntuacion(int puntos, String nombre, long fecha) {
        Uri uri = Uri.parse(
            "content://org.example.puntuacionesprovider/puntuaciones");
        ContentValues valores = new ContentValues();
        valores.put("nombre", nombre);
        valores.put("puntos", puntos);
        valores.put("fecha", fecha);
        try {
            activity.getContentResolver().insert(uri, valores);
        } catch (Exception e) {
            Toast.makeText(activity, "Verificar que está instalado "+
                "org.example.puntuacionesprovider", Toast.LENGTH_LONG).show();
            Log.e("Asteroides", "Error: " + e.toString(), e);
        }
    }
}
```

```
public List<String> listaPuntuaciones(int cantidad) {
    List<String> result = new ArrayList<String>();
    Uri uri = Uri.parse(
        "content://org.example.puntuacionesprovider/puntuaciones");
    Cursor cursor = activity.getContentResolver().query (uri,
        null, null, null, "fecha DESC");

    if (cursor != null) {
        while (cursor.moveToNext()) {
            String nombre = cursor.getString(
                cursor.getColumnIndex("nombre"));
            int puntos = cursor.getInt(
                cursor.getColumnIndex("puntos"));
            result.add(puntos + " " + nombre);
        }
    }
    return result;
}
```

Modifica el código correspondiente para que la nueva clase pueda ser seleccionada como almacén de las puntuaciones. Recuerda que has de instalar primero la aplicación `PuntuacionesProvider` para que funcione este ejemplo.



Preguntas de repaso: *ContentProvider*